前端工程师手册

Leo Hui

Published with GitBook



目錄

front-end-database

HTML/CSS 基础知识

罗列一些基础内容。

资料

学习资料可以参考HTML Dog以及Learn HTML & CSS。

详细的文档内容，推荐参考WebPlatform和MDN。

关于属性的兼容性，可以通过Can I Use查询。

HTML常用标签

div

div标签用于组合其他HTML元素，本身无实在意义。常用于页面的布局，比如一个展开式的广告页面框架大致如下：

```
<body>
    <div id="wrap-container">
        <div id="collapsed-container"></div>
        <div id="expanded-container"></div>
    </div>
</body>
```

h1~h6, p, span, strong, em...

此类标签用于设置文本，常见的使用方式是填充段落，比如弹出的legal框文字HTML结构如下:

```
<div id="legal-window">
    <h4>LEGAL</h4>
    <img id="legal-close" src="img/embed/legal-close.png" alt="close window">
    <p>*Requires a system with Intel<sup>&reg;</sup> Turbo Boost Technology. Intel<sup>&reg;</sup> Turbo Boost Technolo
</div>
```



ul, li, ol, dl, dt, dd

此类标签用于设置带有列表内容的，比如导航栏的下拉菜单，多视频的缩略图等：

```
<ul class="nav-tools-list">
    <li>
        <div>
            <img src="shoppingtools-icon-1.png" alt="">
            <span>Build & Price</span>
        </div>
    </li>
    <li>
        <div>
            <img src="shoppingtools-icon-2.png" alt="">
            <span>Incentives & Offers</span>
        </div>
    </li>
    <li>
        <div>
            <img src="shoppingtools-icon-3.png" alt="">
            <span>Request a Local Quote</span>
        </div>
    </li>
    <li>
        <div>
            <img src="shoppingtools-icon-4.png" alt="">
            <span>Search Dealer Inventory</span>
        </div>
    </li>
</ul>
```

form表单相关

页面中涉及到表单时候，需要使用到form相关标签：

```
<form name="frm-sample" class="frm-sample" action="try" method="post">
    <input type="text" class="form-control" placeholder="Name">
    <div id="status-message"></div>
    <div id="sample-captcha"></div>
    <a id="check-is-filled" class="info-btn">Check if visualCaptcha is filled</a>
    <button type="submit" name="submit-bt" class="submit">Submit form</button>
</form>
```

table表格相关

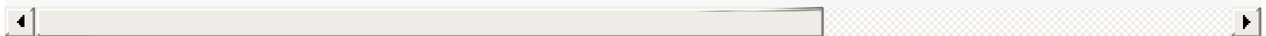
页面中涉及到table结构，需要使用到table相关标签:

```
<talbe></talbe>
```

img, canvas

用于图像显示。一般不直接操作img,canvas元素，而是在它的外层包裹一层父级元素（可以为span,div等)，对父级元素进行操作：

```
<div class="preload" data-src="CheddarBacon.png">
    <img src="CheddarBacon.png" alt="">
</div>
<!-- or -->
<div id="sprite-car" class="cw-sprite sprite-car" cw-interval="30" cw-loops="1" cw-auto-play="false" cw-texture="images
    <canvas class="cw-renderer" width="460" height="130"></canvas>
</div>
```



a

a标签用于打开链接，发送邮件，段落跳转等功能。使用时需要注意阻止掉标签的默认事件。

链接跳转，常见的关于分享按钮的HTML结构如下：

```
<div id="shareBox">
    <ul>
        <li id="facebook">
            <a target="_blank" rel="nofollow" data-shareWay="facebook">
                <img alt="Post on Facebook" src="img/embed/f4Icon3.png" alt="Facebook" />
            </a>
        </li>
        <li id="twitter">
            <a target="_blank" rel="nofollow" data-shareWay="twitter">
                <img alt="Tweet this" src="img/embed/f4Icon4.png" />
            </a>
        </li>
        <li id="pinterest">
            <a data-pin-do="buttonPin" data-pin-config="none" target="_blank" rel="nofollow" data-shareWay="pinterest">
                <img alt="Pin it" src="img/embed/f4Icon5.png" />
            </a>
        </li>
```

```
        <li id="email">
            <a target="_blank" rel="nofollow" data-shareWay="email">
                <img src="img/embed/f4Icon6.png" />
            </a>
        </li>
    </ul>
    <p></p>
</div>
```

发送邮件的代码片段如下：

```
<div class="button">
  <a class="mail" data-img="mail.png" href="mailto:example@gmail.com?subject=xxx&body=xxx"></a>
</div>
```

段落跳转代码片段如下：

```
<div id="html5"></div>
<a name="user-content-html5" href="#html5" class="headeranchor-link" aria-hidden="true"><span class="headeranchor"></sp
```

HTML5标签查询

W3School: 点击查询

HTML 5 NEW TAG

TAG NOT SUPPORTED IN HTML 5

Tag	Description	Attributes
<!--...-->	Define a comment	
<!DOCTYPE>	Defines the document type	
<a>	Defines a hyperlink	href, hreflang, media, ping , rel, target, type
<abbr>	Defines an abbreviation	
<acronym>	Used to define an embedded acronyms	
<address>	Defines an address element	
<applet>	Used to define an embedded applet	
<area>	Defines an area inside an image map	alt, coords, href, hreflang, media, ping, rel, shape, target, type
<article>	Defines an article	cite, pubdate
<aside>	Defines content aside from the page content	
<audio>	Defines sound content	autobuffer, autoplay, controls, src
<b>	Defines bold text	
<base>	Defines a base URL for all the links in a page	href, target
<basefont>	Used to define a default font-color, font-size, or font-family for all the document	
<bdo>	Defines the direction of text display	dir
<big>	Used to make text bigger	
<blockquote>	Defines a long quotation	cite
<body>	Defines the body element	
 	Inserts a single line break	
<button>	Defines a push button	autofocus, disabled, form, formaction, formenctype, formmethod, formnovalidate, formtarget, name, type, value
<canvas>	Defines graphics	height, width
<caption>	Defines a table caption	
<center>	Used to center align text and content	
<cite>	Defines a citation	
<code>	Defines computer code text	autobuffer, autoplay, controls, src
<col>	Defines attributes for table columns	
<colgroup>	Defines groups of table columns	span
<command>	Defines a command button	checked, disabled, icon, label, radiogroup, type
<datalist>	Defines a dropdown list	
<dd>	Defines a definition description	
<del>	Defines deleted text	cite, datetime
<details>	Defines details of an element	open
<dialog>	Defines a dialog (conversation)	
<dfn>	Defines a definition term	
<dir>	Used to define a directory list	
<div>	Defines a section in a document	
<dl>	Defines a definition list	
<dt>	Defines a definition term	
<em>	Defines emphasized text	
<embed>	Defines external interactive content or plugin	height, src, type, width
<fieldset>	Defines a fieldset	disabled, form, name
<figure>	Defines a group of media content, and their caption	
<font>	Used to define font face, font size, and font color of text	
<footer>	Defines a footer for a section or page	
<form>	Defines a form	accept-charset, action, autocomplete, enctype, method, name, novalidate, target
<frame>	Used to define one particular window (frame) within a frameset	
<frameset>	Used to define a frameset, which organized multiple windows (frames)	
<h1> to <h6>	Defines header 1 to header 6	
<head>	Defines information about the document	
<header>	Defines a header for a section or page	
<hgroup>	Defines information about a section in a document	
<hr>	Defines a horizontal rule	
<html>	Defines an html document	manifest, xmlns
<i>	Defines italic text	
<iframe>	Defines an inline sub window	height, name, sandbox, seamless, src, width
<img>	Defines an image	alt, src, height, ismap, usemap, width
<input>	Defines an input field	accept, alt, autocomplete, autofocus, checked, disabled, form, formaction, formenctype, formmethod, formnovalidate, formtarget, height, list, max, maxlength, min, multiple, name, pattern, placeholder, readonly, required, size, src, step, type, value, width
<ins>	Defines inserted text	cite, datetime
<keygen>	Defines a generated key in a form	autofocus, challenge, disabled, form, keytype, name
<kbd>	Defines keyboard text	
<label>	Defines an inline sub window	for, form
<legend>	Defines a title in a fieldset	
	Defines a list item	value
<link>	Defines a resource reference	href, hreflang, media, rel, sizes, type
<map>	Defines an image map	name
<mark>	Defines marked text	
<menu>	Defines a menu list	label, type
<meta>	Defines meta information	charset, content, http-equiv, name
<meter>	Defines measurement within a predefined range	high, low, max, min, optimum, value
<nav>	Defines navigation links	
<noframes>	Used to display text for browsers that do not handle frames	
<noscript>	Defines a noscript section	
<object>	Defines an embedded object	data, form, height, name, type, usemap, width
	Defines an ordered list	reversed, start
<optgroup>	Defines an option group	label, disabled
<option>	Defines an option in a drop-down list	disabled, label, selected, value
<output>	Defines some types of output	for, form, name
<p>	Defines a paragraph	
<param>	Defines a parameter for an object	name, value
<pre>	Defines preformatted text	
<progress>	Defines progress of a task of any kind	max, value
<q>	Defines a short quotation	cite
<rp>	Used in ruby annotations to define what to show browsers that to not support the ruby element	
<rt>	Defines explanation to ruby annotations	
<ruby>	Defines ruby annotations	
<s>, <strike>	Used to define strikethrough text.	
<samp>	Defines sample computer code	
<script>	Defines a definition list	async, type charset defer, src
<section>	Defines a section	cite
<select>	Defines a selectable list	autofocus, disabled, form, multiple, name, size
<small>	Defines small text	
<source>	Defines media resources	media, src, type
<span>	Defines a section in a document	
<strong>	Defines strong text	
<style>	Defines a style definition	type, media, scoped
<sub>, <sup>	Defines sub/super-scripted text	
<table>	Defines a table	summary
<tbody>	Defines a table body	summary
<td>	Defines a table cell	colspan, headers, rowspan
<textarea>	Defines a text area	autofocus, cols, disabled, form, maxlength, name, placeholder, readonly, readonly, required, rows, wrap
<tfoot>, <thead>	Defines a table footer / head	
<th>	Defines a table header	colspan, headers, rowspan, scope
<time>	Defines a date/tim	datetime
<title>	Defines the document title	
<tr>	Defines a table row	datetime
<tt>	Used to define teletype text	
<u>	Used to define underlined text	
	Defines an unordered list	
<var>	Defines a variable	
<video>	Defines a video	autobuffer, autoplay, controls, height, loop, src, width

HTML5 TAG CHEAT SHEET

Created by WebsiteSetup.org

HTML语义化

语义化的含义就是用正确的标签做正确的事情，html语义化就是让页面的内容结构化，便于对浏览器、搜索引擎解析；在没有样式CCS情况下也以一种文档格式显示，并且是容易阅读的。搜索引擎的爬虫依赖于标记来确定上下文和各个关键字的权重，利于 SEO。使阅读源代码的人对网站更容易将网站分块，便于阅读维护理解。

参考资料

- semantic-html
- 关于语义化 HTML 以及前端架构的一点思考
- 如何理解 web 语义化

DOCTYPE和浏览器渲染模式

文档类型，一个文档类型标记是一种标准通用标记语言的文档类型声明，它的目的是要告诉标准通用标记语言解析器，它应该使用什么样的文档类型定义（DTD）来解析文档。Doctype还会对浏览器的渲染模式产生影响，不同的渲染模式会影响到浏览器对于 CSS 代码甚至 JavaScript 脚本的解析，所以Doctype是非常关键的，尤其是在 IE 系列浏览器中，由DOCTYPE 所决定的 HTML 页面的渲染模式至关重要。

浏览器解析HTML方式

有三种解析方式:

- 非怪异（标准）模式
- 怪异模式
- 部分怪异（近乎标准）模式

在“标准模式”(standards mode) 页面按照 HTML 与 CSS 的定义渲染，而在“怪异模式(quirks mode) 模式”中则尝试模拟更旧的浏览器的行为。 一些浏览器（例如，那些基于 Mozilla 的 Gecko 渲染引擎的，或者 Internet Explorer 8 在 strict mode 下）也使用一种尝试于这两者之间妥协的“近乎标准”(almost standards) 模式，实施了一种表单元格尺寸的怪异行为，除此之外符合标准定义。

一个不含任何 DOCTYPE 的网页将会以 怪异(quirks) 模式渲染。

HTML5提供的 `<DOCTYPE html>` 是标准模式，向后兼容的, 等同于开启了标准模式，那么浏览器就得老老实实的按照W3C的 标准解析渲染页面，这样一来，你的页面在所有的浏览器里显示的就都是一个样子了。

参考资料

- DOCTYPE和浏览器渲染模式
- CS002: DOCTYPE 与浏览器模式分析
- 怪异模式（Quirks Mode）对 HTML 页面的影响

理解DOM结构

DOM: Document Object Module, 文档对象模型。我们通过JavaScript操作页面的元素，进行添加、移动、改变或移除的方法和属性, 都是DOM提供的。

W3C DOM 标准

被分为 3 个不同的部分:

- 核心 DOM - 针对任何结构化文档的标准模型
- XML DOM - 针对 XML 文档的标准模型
- HTML DOM - 针对 HTML 文档的标准模型

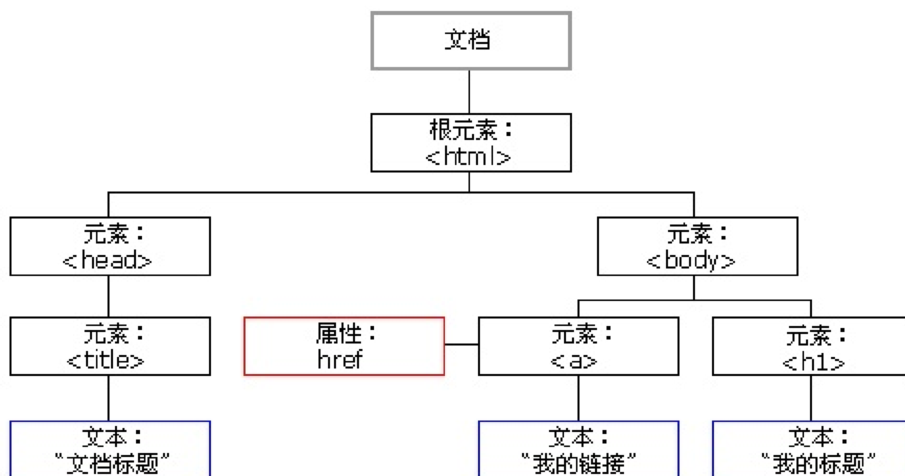
DOM节点

根据 W3C 的 HTML DOM 标准，HTML 文档中的所有内容都是节点：

- 整个文档是一个文档节点
- 每个 HTML 元素是元素节点
- HTML 元素内的文本是文本节点
- 每个 HTML 属性是属性节点
- 注释是注释节点

HTML DOM 节点树

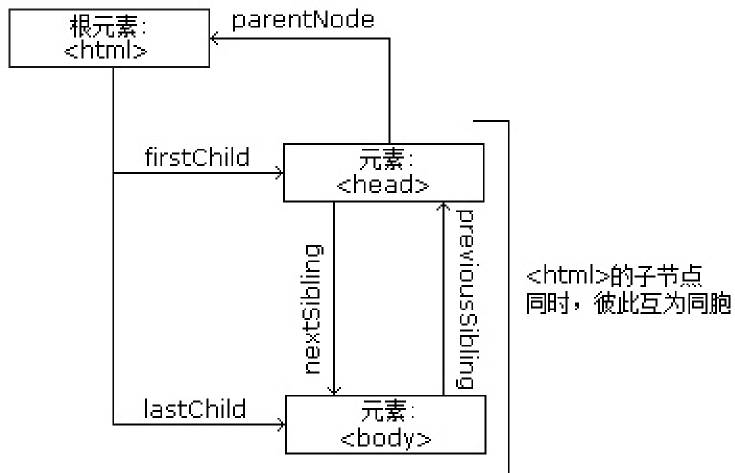
HTML文本会被解析为DOM树, 树中的所有节点均可通过 JavaScript 进行访问。所有 HTML 元素（节点）均可被修改，也可以创建或删除节点。



节点的关系

父（parent）、子（child）和同胞（sibling）等术语用于描述这些关系。父节点拥有子节点。同级的子节点被称为同胞（兄弟或姐妹）：

- 在节点树中，顶端节点被称为根（root）
- 每个节点都有父节点、除了根（它没有父节点）
- 一个节点可拥有任意数量的子
- 同胞是拥有相同父节点的节点



参考资料

- W3C: Document Object Model (DOM) Technical Reports
- MDN: DOM API
- 浏览器的工作原理：新式网络浏览器幕后揭秘
- 开发者需要了解的WebKit
- 理解WebKit和Chromium: HTML解析和DOM
- HTML解析原理

HTML5新增内容

HTML5 是对 HTML 标准的第五次修订。其主要的目标是将互联网语义化，以便更好地被人类和机器阅读，并同时提供更好地支持各种媒体的嵌入。HTML5 的语法是向后兼容的。现在国内普遍说的 H5 是包括了 CSS3，JavaScript 的说法（严格意义上说，这么叫并不合适，但是已经这么叫开了，就将错就错了）。

与HTML 4的不同之处

- 文件类型声明（<!DOCTYPE>）仅有一型：<!DOCTYPE HTML>。
- 新的解析顺序：不再基于SGML。
- 新的元素：section, video, progress, nav, meter, time, aside, canvas, command, datalist, details, embed, figcaption, figure, footer, header, hgroup, keygen, mark, output, rp, rt, ruby, source, summary, wbr。
- input元素的新类型：date, email, url等等。
- 新的属性：ping（用于a与area）, charset（用于meta）, async（用于script）。
- 全域属性：id, tabindex, repeat。
- 新的全域属性：contenteditable, contextmenu, draggable, dropzone, hidden, spellcheck。
- 移除元素：acronym, applet, basefont, big, center, dir, font, frame, frameset, isindex, noframes, strike, tt。

新增标签

HTML 5提供了一些新的元素和属性，反映典型的现代用法网站。其中有些是技术上类似 `<div>` 和 `<span>` 标签，但有一定含义，例如 `<nav>` （网站导航块）和 ``<footer>``<audio>`` 和 `<video>` 标记。

移除的标签

一些过时的HTML 4标记将取消，其中包括纯粹用作显示效果的标记，如 `<font>` 和 `<center>` ，因为它们已经被CSS取代。还有一些通过DOM的网络行为。

修改的标签

尽管和SGML在标记上的相似性，HTML5的句法并不再基于它了，而是被设计成向后兼容对老版本的HTML的解析。它有一个新的开始列看起来就像SGML的文档类型声明， `<!DOCTYPE HTML>` ，这会触发和标准兼容的渲染模式。在2009年1月5号，HTML5添加了Web Form 2.0的内容，html5开始发展起来。

无障碍（Accessibility）

为了使HTML5的新元素或新属性获取最大化的兼容性，开发人员需要附加一点额外补助，或者有些特性根本没有被任何浏览器实现，或者浏览器根本不支持补助技术。因此有些特殊的HTML5特性根本不能使用。更多细节可参见HTML5 Accessibility（无障碍）

新应用程序接口（API）

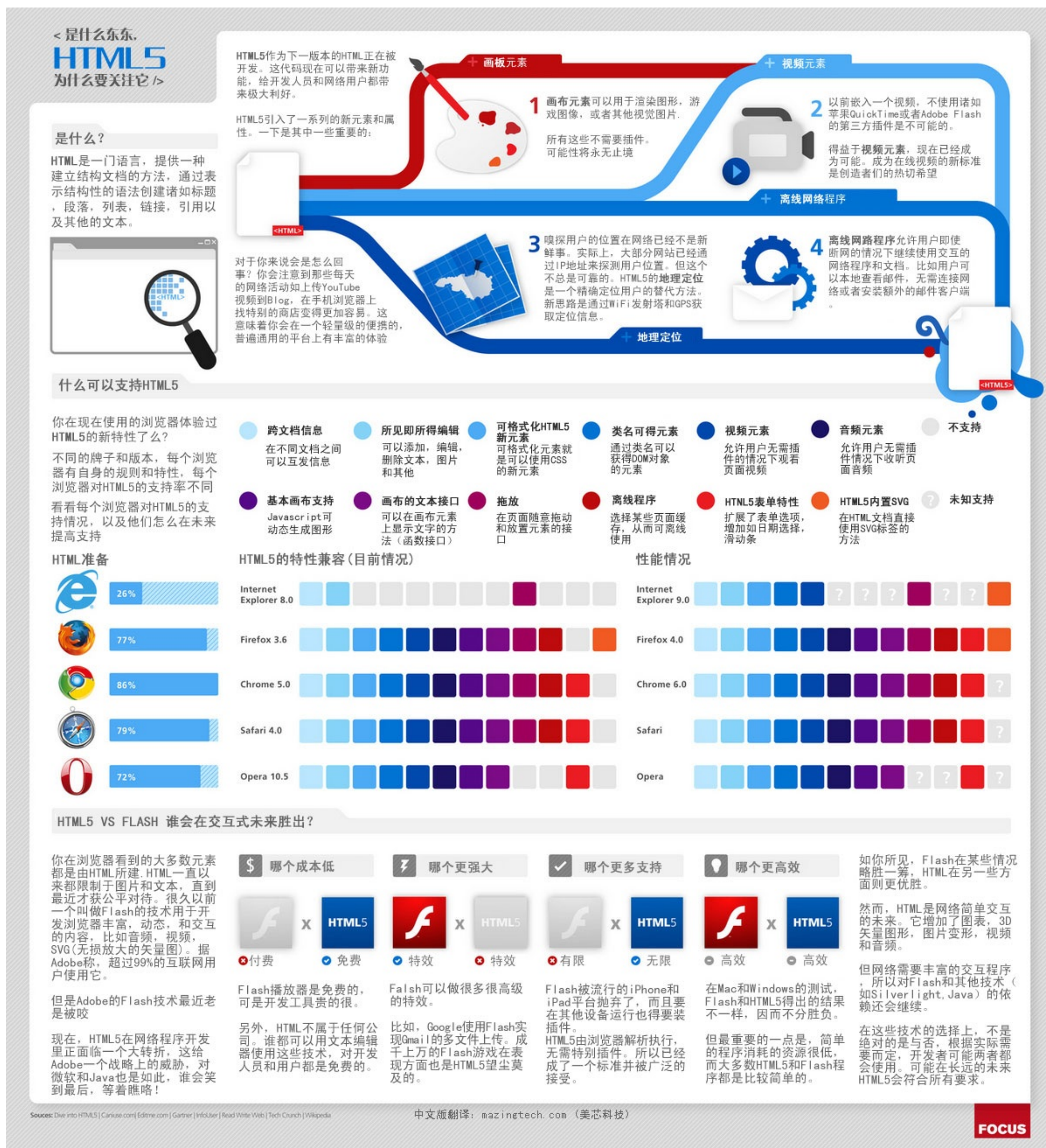
除了原先的DOM接口，HTML5增加了更多样化的API:

- HTML Geolocation
- HTML Drag and Drop

- HTML Local Storage
- HTML Application Cache
- HTML Web Workers
- HTML SSE
- HTML Canvas/WebGL
- HTML Audio/Video

总结

最后奉上一张图片：



参考资料

- 维基百科: html5
- MDN: HTML5
- W3Schools: Learn HTML5
- 知乎: HTML5 到底是什么？

常用CSS属性

常用的CSS属性列表汇总

编辑制作：慕课网

慕课网微信号：imooc-com

近期教学给学员总结常用的CSS属性，方便学习查询，正好发上来也给大家分享一下。

表格最右列的数字标识支持的CSS最低版本。

01. CSS背景属性(Background)

属性与描述

background：在一个声明中设置所有的背景插件1

background-attachment：设置背景图像是否固定或者随着页面的其余部分滚动1

background-color：设置元素的背景颜色1

background-image：设置元素的背景图像1

background-position：设置背景图像的开始位置1

background-repeat：设置是否及如何重复背景图像

02. CSS边框属性(Border和Outline)

属性与描述

border：在一个声明中设置所有的边框属性1

border-bottom：在一个声明中设置所有的下边框属性1

border-bottom-color：设置下边框的颜色2

border-bottom-style：设置下边框的样式2

border-bottom-width：设置下边框的宽度1

border-color：设置四条边框的颜色1

border-left：在一个声明中设置所有的左边框属性1

border-left-color：设置左边框的颜色2

border-left-style：设置左边框的样式2

border-left-width：设置左边框的宽度1

border-right：在一个声明中设置所有右边框的属性1

border-right-color：设置右边框的颜色2

border-right-style：设置右边框的样式2

border-right-width：设置右边框的宽度1

border-style：设置四条边框的样式1

border-top：在一个声明中设置所有上边框的属性1

border-top-color：设置上边框的颜色2

border-top-style：设置上边框的样式2

border-top-width：设置上边框的宽度1

border-width：设置四条边框的宽度1

outline：在一个声明中设置所有的轮廓属性2

outline-color：设置轮廓的颜色2

outline-style：设置轮廓的样式2

outline-width：设置轮廓的宽度

03. CSS文本属性(Text)

属性与描述

color：设置文本的颜色1

direction：规定文本的方向/书写方向2

letter-spacing：设置字符间距1

line-height：设置行高1

text-align：规定文本的水平对齐方式1

text-decoration：规定添加到文本的装饰效果1

text-indent：规定文本块首行的缩进1

text-shadow：规定添加到文本的阴影效果2

text-transform：控制文本的大小写1

unicode-bidi：设置文本方向2

white-space：规定如何处理元素中的空白1

word-spacing：设置单词间距1

04. CSS字体属性(Font)

属性与描述

font：在一个声明中设置所有字体属性1

font-family：规定文本的字体系列1

font-size：规定文本的字体尺寸1

font-size-adjust：为元素规定aspect值2

font-stretch：收缩或拉伸当前的字体系列2

font-style：规定文本的字体压实1

font-variant：规定文本的字体样式1

font-weight：规定字体的粗细

05. CSS外边距属性(Margin)

属性与描述

margin：在一个声明中设置所有的外边距属性1

margin-bottom:设置元素的下外边距1

margin-left:设置元素的左外边距1

margin-right:设置元素的右外边距1

margin-top:设置元素的上外边距

06. CSS内边距属性(Padding)

属性与描述

padding:在一个声明中设置所有的内边距属性1

padding-bottom:设置元素的下内边距1

padding-left:设置元素的左内边距1

padding-right:设置元素的右内边距1

padding-top:设置元素的上内边距

07. CSS列表属性(List)

属性与描述

list-style:在一个声明中设置所有的列表属性1

list-style-image:将图像设置为列表项标记1

list-position:设置列表项标记的放置位置1

list-style-type:设置列表项标记的类型

08. CSS尺寸属性(Dimension)

属性与描述

height:设置元素高度1

max-height:设置元素的最大高度2

max-width:设置元素的最大宽度2

min-height:设置元素的最小高度2

min-width:设置元素的最小宽度2

width:设置元素的宽度1

09. CSS定位属性(Positioning)

属性与描述

bottom:设置定位元素下外边距边界与其包含块下边界之间的偏移2

clear:规定元素的哪一侧不允许其他浮动元素1

clip:剪裁绝对定位元素2

cursor:规定要显示的光标类型(形状)2

display:规定元素应该生成的框的类型1

float:规定框是否应该浮动1

left:设置定位元素左外边距边界与其包含块左边界之间的偏移2

overflow:规定当内容溢出元素框时发生的事情2

position:规定元素定位类型2

right:设置定位元素右外边距边界与其包含块右边之间的偏移2

top:设置定位元素上外边距边界与其包含块上边之间的偏移2

vertical-align:设置元素的垂直对其方式1

visibility:规定元素是否可见2

z-index:设置元素的堆叠顺序

10. CSS表格属性(Table)

属性与描述

border-collapse:规定是否合并表格边框2

border-spacing:规定相邻单元格边框之间的距离2

caption-side:规定表格标题的位置2

empty-cells:规定是否显示表格中的空单元格上的边框和背景2

table-layout:设置用于表格的布局算法

文章源自：w3cschool

CSS普通流（文档流）

首先明确一点的是，W3C规范中没有 `document flow` 这个概念，只有normal-flow, 文档流的叫法主要还是多数中文译者的翻译方式问题。

定义

什么是普通流？简单说就是元素按照其在 HTML 中的位置顺序决定排布的过程。并且这种过程遵循标准的描述。

参考资料

- normal-flow
- CSS定位机制之一：普通流

CSS定位方式

参考资料

- CSS 相对|绝对(relative/absolute)定位系列（一）
- CSS 相对/绝对(relative/absolute)定位系列（二）
- CSS 相对/绝对(relative/absolute)定位系列（三）
- CSS 相对/绝对(relative/absolute)定位系列（四）
- CSS相对定位|绝对定位(五)之z-index篇

CSS选择器

选择器兼容性

	iPhn		Windows XP											Mac OSX							
Selector	Saf 3.2	Saf 3.0	goog chrm	FF 3.5	FF 3.0	FF 2.0	FF 1.5	Op 9.0	Saf 3.0	IE8	IE7 in IE8	IE7	IE6	Saf 4 beta	Saf 3.2	Saf 3.1	Saf 1.3	Op 9.64	FF 3	FF 2	NS 7.1
*	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
E	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
.class	√	√	√	√	√	√	√	√	√	√	√	√	△	√	√	√	√	√	√	√	√
#id	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
E F	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
E > F	√	√	√	√	√	√	√	√	√	√	√	√	×	√	√	√	√	√	√	√	√
E + F	√	√	√	√	√	√	√	√	√	√	√	√	×	√	√	√	√	√	√	√	√
E[attr]	√	√	√	√	√	√	√	√	√	△	△	△	×	√	√	√	√	√	√	√	△
E[attr=val]	√	△	√	√	√	△	△	△	△	√	√	√	×	√	√	√	△	√	√	△	△
E[attr~=val]	√	△	√	√	√	△	△	√	△	√	△	△	×	√	√	√	△	√	√	△	△
E[attr|=val]	√	△	√	√	√	△	△	√	△	√	△	△	×	√	√	√	△	√	√	△	△
:first-child	√	△	√	√	√	△	△	△	△	√	△	△	×	√	√	√	△	√	√	△	√
:link	√	√	√	√	√	√	√	×	√	√	√	√	√	√	√	√	√	√	√	√	√
:visited	√	√	√	√	√	√	√	×	√	√	√	√	√	√	√	√	√	√	√	√	√
:lang()	√	√	√	√	√	√	√	△	√	√	×	×	×	√	√	√	×	√	√	√	√
:before	√	√	√	√	√	√	√	√	√	√	×	×	×	√	√	√	√	√	√	√	×
::before	√	√	√	√	√	√	√	√	√	×	×	×	×	√	√	√	√	√	√	√	×
:after	√	√	√	√	√	√	√	√	√	√	×	×	×	√	√	√	√	√	√	√	×
::after	√	√	√	√	√	√	√	√	√	×	×	×	×	√	√	√	√	√	√	√	×
:first-letter	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	×
::first-letter	√	√	√	√	√	√	√	√	√	×	×	×	√	√	√	√	√	√	√	√	×
:first-line	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	×
::first-line	√	√	√	√	√	√	√	√	√	×	×	×	√	√	√	√	√	√	√	√	×
The following selectors are new to CSS3 (above were in previous versions)																					
E[attr^=val]	√	△	√	√	√	△	△	×	△	√	×	×	×	√	√	√	△	√	√	△	△
E[attr$=val]	√	△	√	√	√	△	△	×	△	√	×	×	×	√	√	√	△	√	√	△	△
E[attr*=val]	√	△	√	√	√	△	△	√	△	√	×	×	×	√	√	√	△	√	√	△	△
E ~ F	√	√	√	√	√	√	√	√	√	√	√	√	×	√	√	√	×	√	√	√	×
:root	√	√	√	√	√	√	√	×	√	×	×	×	×	√	√	√	√	√	√	√	√
:last-child	√	×	√	√	√	△	△	×	×	×	×	×	×	√	√	√	×	√	√	△	△
:only-child	√	×	√	√	√	△	△	×	×	×	×	×	×	√	√	√	×	√	√	△	×
:nth-child()	√	×	√	√	×	×	×	×	×	×	×	×	×	√	√	√	×	√	×	×	×
:nth-last-child()	√	×	√	√	×	×	×	×	×	×	×	×	×	√	√	√	×	√	×	×	×
:first-of-type	√	△	√	√	×	×	×	×	△	×	×	×	×	√	√	√	×	√	×	×	×
:last-of-type	√	×	√	√	×	×	×	×	×	×	×	×	×	√	√	√	×	√	×	×	×
:only-of-type	√	×	√	√	×	×	×	×	×	×	×	×	×	√	√	√	×	√	×	×	×
:nth-of-type()	√	×	√	√	×	×	×	×	×	×	×	×	×	√	√	√	×	√	×	×	×
:nth-last-of-type()	√	×	√	√	×	×	×	×	×	×	×	×	×	√	√	√	×	√	×	×	×
:empty	√	×	√	√	√	△	△	×	×	×	×	×	×	√	√	√	×	√	√	△	△
:not()	√	√	√	√	√	√	√	×	√	×	×	×	×	√	√	√	√	√	√	√	√
:target	√	√	√	√	√	√	√	×	√	×	×	×	×	√	√	√	√	√	√	√	√
:enabled	√	√	√	√	√	√	√	√	√	×	×	×	×	√	√	√	×	√	√	√	×
:disabled	√	√	√	√	√	√	√	√	√	×	×	×	×	√	√	√	×	√	√	√	×
:checked	√	√	√	√	√	√	√	√	√	×	×	×	×	√	√	√	×	√	√	√	×

常用meta整理

概要

标签提供关于HTML文档的元数据。元数据不会显示在页面上，但是对于机器是可读的。它可用于浏览器（如何显示内容或重新加载页面），搜索引擎（关键词），或其他 web 服务。

基本属性

属性	值	描述
keywords	author / description / keywords / generator / revised / others	把 content 属性关联到一个名称。
content	some text	定义与http-equiv或name属性相关的元信息
http-equiv	content-type / expire / refresh / set-cookie	把content属性关联到HTTP头部。

参考资料

- 常用meta整理

什么是Viewport

参考资料

- 什么是Viewport Meta（width详解）及在手机上的应用

CSS动画

参考资料

- css3动画总结

CSS基线

基线是排版上的一种名词，是指大部分字母所“坐落”其上的一条看不见的线，每条基线之间形成基本的基线网格。但是在web中，当我们需要垂直对齐的时候，基线往往让人失望，比如`line-height`属性。

参考资料

- Wikipedia: baseline)
- CSS基线之道

CSS动画

参考资料

- 利用 CSS animation 和 CSS sprite 制作动画

前端UI框架

Web Component是未来的一种趋势，现阶段，还是很多CSS的基础框架，比如React。

参考资料

- 颠覆式前端UI开发框架：React

JavaScript基础知识

描述一些容易混淆或者忘记的知识点。比如mouse事件的顺序，event属性有哪些。。。

参考资料

- 如何正确学习JavaScript？
- JavaScript 标准参考教程（alpha）:阮一峰写的教程。
- JavaScript简易教程：最完整最简洁的JavaScript基础教程。

作用域问题

JavaScript语言的作用域仅存在于函数范围中。这是必须要牢记的一点，还有一点重要的就是作用域的提升规则。

作用域问题

JS最容易出现混淆的就是作用域的情况。传统的类C语言,它们的作用域是`block-level scope`，块级作用域，花括号就是一个作用域。但是对于JavaScript而言，它的作用域是`function-level scope`，比如if条件语句，就不算一个独立的作用域:

```
var x = 1;
console.log(x); // 1
if (true) {
    var x = 2;
    console.log(x); // 2
}
console.log(x); // 2
```

在JavaScript中，如果我们需要实现`block-level scope`，我们也有一种变通的方式，那就是通过自执行函数创建临时作用域:

```
function foo() {
    var x = 1;
    if (x) {
        (function () {
            var x = 2;
            // some other code
        }());
    }
    // x is still 1.
}
```

作用域提升

变量被提升

对JavaScript解释器而言，所有的函数和变量声明都会被提升到最前面, 并且变量声明永远在前面，赋值在声明过程之后。比如:

```
var x = 10;
function x(){};
console.log(x); // 10
```

实际上被解释为:

```
var x;
function x(){};
x = 10;
console.log(x); // 10
```

函数被提升

函数的声明方式主要由两种：声明式和变量式。

声明式会自动将声明放在前面并且执行赋值过程。而变量式则是先将声明提升，然后到赋值处再执行赋值。比如:

```
function test() {
    foo(); // TypeError "foo is not a function"
    bar(); // "this will run!"
    var foo = function () { // function expression assigned to local variable 'foo'
        alert("this won't run!");
    }
    function bar() { // function declaration, given the name 'bar'
        alert("this will run!");
    }
}
test();
```

实际上等价于:

```
function test() {
    var foo;
    var bar;
    bar = function () { // function declaration, given the name 'bar'
        alert("this will run!");
    }

    foo(); // TypeError "foo is not a function"
    bar(); // "this will run!"

    foo = function () { // function expression assigned to local variable 'foo'
        alert("this won't run!");
    }
}
test();
```

主要注意的地方：带有命名的函数变量式声明，是不会提升到作用域范围内的，比如:

```
var baz = function spam() {};
baz(); // vaild
spam(); // ReferenceError "spam is not defined"
```

实战经验

任何时候，请使用 **`var`** 声明变量, 并放置在作用域的顶端.

工具推荐 `JSLint` 之类，帮助你验证语法的规范。

参考资料

- JavaScript Scoping and Hoisting
- ECMScript Standard(PDF)

运算符的优先级

任何一门语言都会遵守这样的规则。

参考资料

- MDN: 运算符优先级
- JS Comparison Table: if, ==, ===各种对比情况
- 代码之谜（三）- 运算符

undefined与null的区别

在常见的强类型语言中，通常有一个表示“空”的值，比如NULL。但是在Javascript中，空（或者叫“无值”）有两种选择：undefined和null。在Javascript中除了这两个值其他都是对象。其他的基本类型都有其对象的包装类型。但是，`typeof null`返回的是`object`，这是一个一直未修复的bug。

相似之处

都是完全不可变的，没有属性和方法，也不能给其属性赋值。事实上,试图访问或定义一个属性将会引发一个类型错误（TypeError）。正如他们的名字暗示的那样，他们是完全无效的值。

不同之处

一个重要的区别，服务于不同的目的和理由。区分这两个值，你可以认为undefined代表一个意想不到的没有值而null作为预期没有值的代表。

使用`Object.prototype.toString.call()`形式可以具体打印类型。

undefined

undefined实际上代表了不存在的值（non-existence of a value）。

有许多的方法产生一个undefined值的代码。它通常遇到当试图访问一个不存在的值时。在这种情况下，在JavaScript这种动态的弱类型语言中，只会默认返回一个undefined值，而不是上升为一个错误:

- 任何声明变量时没有提供一个初始值，都会有一个为undefined的默认值
- 当试图访问一个不存在的对象属性或数组项时，返回一个undefined值
- 如果省略了函数的返回语句,返回undefined
- 函数调用时未提供的值结果将为undefined参数值
- void操作符也可以返回一个undefined值。像Underscore的库使用它作为一个防御式的类型检查，因为它是不可变的，可以在任何上下文依赖返回undefined
- undefined是一个预定义的全局变量(不像null关键字)初始化为undefined值

null

通常用作一个空引用一个空对象的预期,就像一个占位符。typeof的这种行为已经被确认为一个错误，虽然提出了修正，出于后兼容的目的，这一点已经保持不变。 这就是为什么JavaScript环境从来没有设置一个值为null；它必须以编程方式完成。

使用null的情况:

- DOM，它是独立于语言的，不属于ECMAScript规范的范围。因为它是一个外部API，试图获取一个不存在的元素返回一个null值，而不是undefined。
- 如果你需要给一个变量或属性指定一个不变值，将它传递给一个函数，或者从一个函数返回null，null几乎总是最好的选择。
- JavaScript使用undefined并且程序员应该使用null。
- 通过分配null值，有效地清除引用，并假设对象没有引用其他代码，指定垃圾收集，确保回收内存。

Object.prototype.toString调用过程

- 如果值是undefined，返回“[object Undefined]”。
- 如果这个值为null，则返回“[object Null]”。
- 让O作为调用ToObject同时传递this值作为参数的结果值。
- 让class是O的内部属性[[Class]]的值。
- 返回的结果连接三个字符串“[object ”，class，和“]”的结果的字符串值。

参考资料

- 探索JavaScript中Null和Undefined的深渊

内置对象与原生对象

参考资料

- JavaScript原生对象及扩展

关于函数

理解函数的几种声明方式，以及其中的差异。

参考资料

- 在JavaScript的Array数组中调用一组Function方法

事件机制

JavaScript是一套使用事件机制较多的语言，特别是与DOM交互的时候。所以了解并理解事件机制就变得很必要了。

事件监听

HTML内联属性

类似 `<button onclick="alert('你点击了这个按钮');">点击这个按钮</button>` 的方式，这种方式会使JS与HTML高度耦合，不利于开发和维护，不推荐使用。

DOM属性绑定

使用DOM元素的 `onXXX` 属性设置，简单易懂，兼容性好。确定是只能绑定一个处理函数。

事件监听函数

使用事件监听函数 `element.addEventListener(<event-name>, <callback>, <use-capture>);` ，在 element 这个对象上面添加一个事件监听器，当监听到有 事件发生的时候，调用 这个回调函数。至于 这个参数，表示该事件监听是在“捕获”阶段中监听（设置为 true）还是在“冒泡”阶段中监听（设置为 false）。

移除事件监听

使用事件解除绑定方法: `element.removeEventListener(<event-name>, <callback>, <use-capture>);`

需要注意的是，绑定事件时的回调函数不能是匿名函数，必须是一个声明的函数，因为解除事件绑定时需要传递这个回调函数的引用，才可以断开绑定。

模拟触发事件

内置的时间也可以被JavaScript模拟触发，使用 `dispatchEvent` 方法。

自定义事件

与自定义事件的函数有 Event、CustomEvent 和 dispatchEvent。

Event

直接自定义事件，使用 Event 构造函数：

```
var event = new Event('build');

// Listen for the event.
elem.addEventListener('build', function (e) { ... }, false);

// Dispatch the event.
elem.dispatchEvent(event);
```

CustonEvent

CustomEvent 可以创建一个更高度自定义事件，还可以附带一些数据，具体用法如下：

```
var myEvent = new CustomEvent(eventname, options);
```

其中options可以是:

```
{
    detail: {
        ...
    },
    bubbles: true,
    cancelable: false
}
```

其中 detail 可以存放一些初始化的信息，可以在触发的时候调用。其他属性就是定义该事件是否具有冒泡等等功能。

dispatchEvent

这个用于触发自定义的事件。

事件顺序

当我们给父子关系的元素都绑定了事件的时候，触发子元素的时候，这两个事件发生的前后顺序是如何的？这引开了我们关于事件顺序的讨论，其实一共有两种方式:

- Event Capturing(事件捕获)： NetScape所主张的方式
- Event Bubbling(事件冒泡)： Micsoft所主张的方式

这两种方式确定了事件执行的前后顺序，只不过后来W3C对DOM2的事件模型给出了一个规范：首先进入事件捕获阶段->达到元素后->进入事件冒泡阶段。

开发者可以通过 `addEventListener` 函数的第三个参数设置事件触发的阶段，默认为false,冒泡阶段。而DOM1级别的事件绑定则只能在冒泡阶段触发。

事件代理

事件绑定后，检测顺序就会从被绑定的DOM下滑到触发的元素，再冒泡会绑定的DOM上。也就是说，如果你监听了一个DOM节点，那也就等于你监听了其所有的后代节点。

代理的意思就是只监听父节点的事件触发，以来代理对其后代节点的监听，而你需要做的只是通过 `currentTarget` 属性得到触发元素并作出回应。

使用事件代理意味着你可以节省大量重复的事件监听，以减少浏览器资源消耗。还有一个好处就是让HTML独立起来，比如之后还有要加子元素的需求，也不需要再为其单独加事件监听了。

事件的Event对象

当一个事件被触发的时候，会创建一个事件对象（Event Object），这个对象里面包含了一些有用的属性或者方法。事件对象会作为第一个参数，传递给我们的毁掉函数。我们可以使用下面代码，在浏览器中打印出这个事件对象：

```
    var btn = document.getElementsByTagName('button');
    btn[0].addEventListener('click', function(event) {
        console.log(event);
    }, false);
```

比较常用的几个属性和方法：

- type(string): 事件的名称，比如 “click”。
- target(node): 事件要触发的目标节点。
- currentTarget(node): 它就指向正在处理事件的元素：这恰是我们需要的。很不幸的是微软模型中并没有相似的属性, 你也可以使用”this”关键字。事件属性也提供了一个值可供访问: `event.currentTarget` 。
- bubbles (boolean): 表明该事件是否是在冒泡阶段触发的。
- preventDefault (function): 这个方法可以禁止一切默认的行为，例如点击 a 标签时，会打开一个新页面，如果为 a 标签监听事件 click 同时调用该方法，则不会打开新页面。
- stopPropagation (function): 很多时候，我们触发某个元素，会顺带触发出它父级身上的事件，这有时候是我们不想要的，大多数我们想要的还是事件相互独立。所以我们可以选择阻止事件冒泡，使用 `event.stopPropagation()` .
- stopImmediatePropagation (function): 与 stopPropagation 类似，就是阻止触发其他监听函数。但是与 stopPropagation 不同的是，它更加 “强力”，阻止除了目标之外的事件触发，甚至阻止针对同一个目标节点的相同事件
- cancelable (boolean): 这个属性表明该事件是否可以通过调用 event.preventDefault 方法来禁用默认行为。
- eventPhase (number): 这个属性的数字表示当前事件触发在什么阶段。
  - 0: none
  - 1: 捕获
  - 2: 目标
  - 3: 冒泡
- pageX 和 pageY (number): 这两个属性表示触发事件时，鼠标相对于页面的坐标。
- isTrusted (boolean): 表明该事件是浏览器触发（用户真实操作触发），还是 JavaScript 代码触发的。

事件的回调函数

事件绑定函数时，该函数会以当前元素为作用域执行,所以回调函数中的 `this` 是当前的DOM元素。如果我们需要指定作用域，可以选择:

- 使用匿名函数包裹回调函数
- 使用bind方法

事件列表

可以通过MDN查询，也可以在浏览器中输入:

```
for (i in window) {
  if ( /^on/.test(i)) { console.log(i); }
}
```

查看，你会发现提供的事件超过你想象的多！

常用事件

- `load` 资源加载完成时触发。这个资源可以是图片、CSS 文件、JS 文件、视频、document 和 window 等等。
- `DOMContentLoaded` DOM构建完毕的时候触发, jQuery的ready方法包裹的就是这个事件。
- `beforeunload` 当浏览者在页面上的输入框输入一些内容时，未保存、误操作关掉网页可能会导致输入信息丢失。当浏览者输入信息但未保存时关掉网页，我们就可以开始监听这个事件,这时候试图关闭网页的时候，会弹窗阻止操作，点击确

认之后才会关闭。当然，如果没有必要，就不要监听，不要以为使用它可以为你留住浏览者。
- `resize` 当节点尺寸发生变化时，触发这个事件。通常用在 window 上，这样可以监听浏览器窗口的变化。通常用在复杂布局和响应式上。出于对性能的考虑，你可以使用函数 throttle 或者 debounce 技巧来进行优化，throttle 方法大体思路就是在某一段时间内无论多次调用，只执行一次函数，到达时间就执行；debounce 方法大体思路就是在某一段时间内等待是否还会重复调用，如果不会再调用，就执行函数，如果还有重复调用，则不执行继续等待。
- `error` 当我们加载资源失败或者加载成功但是只加载一部分而无法使用时，就会触发 error 事件，我们可以通过监听该事件来提示一个友好的报错或者进行其他处理。比如 JS 资源加载失败，则提示尝试刷新；图片资源加载失败，在图片下面提示图片加载失败等。该事件不会冒泡。因为子节点加载失败，并不意味着父节点加载失败，所以你的处理函数必须精确绑定到目标节点。

IE浏览器下的情况

IE下绑定事件

在 IE 下面绑定一个事件监听，在 IE9之前的版本中无法使用标准的 `addEventListener` 函数，而是使用自家的 `attachEvent`，具体用法：`element.attachEvent(<event-name>, <callback>);`

它只支持监听在冒泡阶段触发的事件，所以为了统一，在使用标准事件监听函数的时候，第三参数传递 false。

IE下的Event事件

IE 中往回调函数中传递的事件对象与标准也有一些差异，你需要使用 window.event 来获取事件对象。所以你通常会写出下面代码来获取事件对象：

```
event = event || window.event
```

此外还有一些事件属性有差别，比如比较常用的 event.target 属性，IE 中没有，而是使用 event.srcElement 来代替。如果你的回调函数需要处理触发事件的节点，那么需要写:

```
node = event.srcElement || event.target;
```

参考资料

- Introduction to Events
- Event order
- MDN: Event
- W3C: Document Object Model (DOM) Level 3 Events Specification
- 你若触发，我就处理——浅谈JavaScript的事件响应
- 最详细的JavaScript和事件解读

原型继承

JavaScript不是真正意义上的面向对象语言，没有提供传统的继承方式,它提供的是一种叫做原型继承的方式。但是它拥有面向对象和函数式的编程特点,理解原型继承，对我们使用JS来实现面向对象很有帮助。

原型与原型链

在Javascript中，每个函数都有一个原型属性prototype指向自身的原型，而由这个函数创建的对象也有一个`__proto__`属性指向这个原型，而函数的原型是一个对象，所以这个对象也会有一个`__proto__`指向自己的原型，这样逐层深入直到Object对象的原型，这样就形成了原型链。

每个函数都是Function函数创建的对象，所以每个函数也有一个`__proto__`属性指向Function函数的原型。这里需要指出的是，真正形成原型链的是每个对象的`__proto__`属性，而不是函数的prototype属性，这是很重要的。

原型继承与类继承的区别

基于类的继承	原型继承
类是不可变的。在运行时，你无法修改或者添加新的方法	原型是灵活的。它们可以是不可变的也可以是可变的
类可能会不支持多重继承	对象可以继承多个原型对象
基于类的继承比较复杂。你需要使用抽象类，接口和final类等等	原型继承比较简洁。你只有对象，你只需要对对象进行扩展就可以了

创建方式

构造模式(使用**new**关键字)

由于JavaScript中所有函数都可以充当对象的构造函数，所以可以通过关键字`new`来进行区分, 但是new并不是一个方法。为了使得Javascript看起来更像Java原型模式被迫屈服于构造模式。因此每个Javascript中的函数都有一个prototype对象然后可以用来作为构造器(这里构造器的意思应该是说新的对象是在prototype对象的基础上进行构造的)。new关键字允许我们把函数当做构造函数使用。它会克隆构造函数的prototype属性然后把它绑定到this对象中，如果没有显式返回对象则会返回this。

```
var Parent = function(){
    this.name = 'parent' ;
} ;
Parent.prototype.getName = function(){
    return this.name ;
} ;
Parent.prototype.obj = {a : 1} ;

var Child = function(){
    this.name = 'child' ;
} ;
Child.prototype = new Parent() ;

var parent = new Parent() ;
var child = new Child() ;
```

原型模式(不要使用**new**关键字)

new关键字掩盖了Javascript中真正的原型继承，使得它更像是基于类的继承。就像Raynos说的:

```
new是Javascript在为了获得流行度而加入与Java类似的语法时期留下来的一个残留物
```

Javascript是一个源于Self的基于原型的语言。然而，为了市场需求，Brendan Eich把它当成Java的小兄弟推出：

```
并且我们当时把Javascript当成Java的一个小兄弟，就像在微软语言家庭中Visual Basic相对于C++一样。
```

这个设计决策导致了new的问题。当人们看到Javascript中的new关键字，他们就想到类，然后当他们使用继承时就遇到了傻了。就像Douglas Crockford说的：

```
这个间接的行为是为了使传统的程序员对这门语言更熟悉，但是却失败了，就像我们看到的很少Java程序员选择了Javascript。Javascript的构造模式并没有
```

因此我建议停止使用new关键字。Javascript在传统面向对象假象下面有着更加强大的原型系统。然大部分程序员并没有看见这些还处于黑暗中。创建对象使用 `Object.create()` 方法。

```
javascriptvar rectangle = {
    create : function(width,height){
        var self = Object.create(this) ;
        self.height = height ;
        self.width = width ;
        return self ;
    } ,
    area : function(){
        return this.width * this.height ;
    }
} ;
var rect = rectangle.create(5,10) ;
alert(rect.area()) ;
```

构造模式和原型模式对比

构造模式	原型模式
函数式特点无法与new关键字一起使用	函数式特点可以与create结合使用
忘记使用new会导致无法预期的bug并且会污染全局变量	由于create是一个函数，所以程序总是会按照预期工作
使用构造函数的原型继承比较复杂并且混乱	使用原型的原型继承简洁易懂

参考资料

- 为什么原型继承很重要
- 再谈Javascript原型继承

详解this

习惯了高级语言的你或许觉得JavaScript中的this跟Java这些面向对象语言相似，保存了实体属性的一些值。其实不然。将它视作幻影魔神比较恰当，手提一个装满未知符文的灵龛。

全局this

浏览器宿主的全局环境中，this指的是window对象。

```
<script type="text/javascript">
    console.log(this === window); //true
</script>
```

浏览器中在全局环境下，使用var声明变量其实就是赋值给this或window。

```
<script type="text/javascript">
    var foo = "bar";
    console.log(this.foo); //logs "bar"
    console.log(window.foo); //logs "bar"
</script>
```

任何情况下，创建变量时没有使用var或者let(ECMAScript 6)，也是在操作全局this。

```
<script type="text/javascript">
    foo = "bar";

    function testThis() {
      foo = "foo";
    }

    console.log(this.foo); //logs "bar"
    testThis();
    console.log(this.foo); //logs "foo"
</script>
```

Node命令行（REPL）中，this是全局命名空间。可以通过global来访问。

```
> this
{ ArrayBuffer: [Function: ArrayBuffer],
  Int8Array: { [Function: Int8Array] BYTES_PER_ELEMENT: 1 },
  Uint8Array: { [Function: Uint8Array] BYTES_PER_ELEMENT: 1 },
  ...
> global === this
true
```

在Node环境里执行的JS脚本中，this其实是个空对象，有别于global。

```
console.log(this);
console.log(this === global);

###
$ node test.js
{}
false
```

但在命令行里进行求值却会赋值到this身上。

```
> var foo = "bar";
> this.foo
bar
> global.foo
bar
```

在Node里执行的脚本中，创建变量时没带var或let关键字，会赋值给全局的global但不是this（译注：上面已经提到this和global不是同一个对象，所以这里就不奇怪了）。

```
foo = "bar";
console.log(this.foo);
console.log(global.foo);

###
$ node test.js
undefined
bar
```

但在Node命令行里，就会赋值给两者了。

简单来说，**Node**脚本中**global**和**this**是区别对待的，而**Node**命令行中，两者可等效为同一对象。

函数或方法里的this

除了DOM的事件回调或者提供了执行上下文（后面会提到）的情况，函数正常被调用（不带new）时，里面的this指向的是全局作用域。

```
<script type="text/javascript">
    foo = "bar";

    function testThis() {
      this.foo = "foo";
    }

    console.log(this.foo); //logs "bar"
    testThis();
    console.log(this.foo); //logs "foo"
</script>
```

测试结果：

```
$ node test.js
bar
foo
```

还有个例外，就是使用了"use strict";。此时this是undefined。

```
<script type="text/javascript">
    foo = "bar";

    function testThis() {
      "use strict";
      this.foo = "foo";
    }

    console.log(this.foo); //logs "bar"
```

```
    testThis();  //Uncaught TypeError: Cannot set property 'foo' of undefined
</script>
```

当用调用函数时使用了new关键字，此刻this指代一个新的上下文，不再指向全局this。

```
<script type="text/javascript">
    foo = "bar";

    function testThis() {
      this.foo = "foo";
    }

    console.log(this.foo); //logs "bar"
    new testThis();
    console.log(this.foo); //logs "bar"

    console.log(new testThis().foo); //logs "foo"
</script>
```

通常我将这个新的上下文称作实例。

原型中的this

函数创建后其实以一个函数对象的形式存在着。既然是对象，则自动获得了一个叫做prototype的属性，可以自由地对这个属性进行赋值。当配合new关键字来调用一个函数创建实例后，此刻便能直接访问到原型身上的值。

```
function Thing() {
    console.log(this.foo);
}

Thing.prototype.foo = "bar";

var thing = new Thing(); //logs "bar"
console.log(thing.foo);  //logs "bar"
```

当通过new的方式创建了多个实例后，他们会共用一个原型。比如，每个实例的this.foo都返回相同的值，直到this.foo被重写。

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    console.log(this.foo);
}
Thing.prototype.setFoo = function (newFoo) {
    this.foo = newFoo;
}

var thing1 = new Thing();
var thing2 = new Thing();

thing1.logFoo(); //logs "bar"
thing2.logFoo(); //logs "bar"

thing1.setFoo("foo");
thing1.logFoo(); //logs "foo";
thing2.logFoo(); //logs "bar";

thing2.foo = "foobar";
thing1.logFoo(); //logs "foo";
thing2.logFoo(); //logs "foobar";
```

在实例中，this是个特殊的对象，而this自身其实只是个关键字。你可以把this想象成在实例中获取原型值的一种途径，同时对this赋值又会覆盖原型上的值。完全可以将新增的值从原型中删除从而将原型还原为初始状态。

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    console.log(this.foo);
}
Thing.prototype.setFoo = function (newFoo) {
    this.foo = newFoo;
}
Thing.prototype.deleteFoo = function () {
    delete this.foo;
}

var thing = new Thing();
thing.setFoo("foo");
thing.logFoo(); //logs "foo";
thing.deleteFoo();
thing.logFoo(); //logs "bar";
thing.foo = "foobar";
thing.logFoo(); //logs "foobar";
delete thing.foo;
thing.logFoo(); //logs "bar";
```

或者不通过实例，直接操作函数的原型。

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    console.log(this.foo, Thing.prototype.foo);
}

var thing = new Thing();
thing.foo = "foo";
thing.logFoo(); //logs "foo bar";
```

同一函数创建的所有实例均共享一个原型。如果你给原型赋值了一个数组，那么所有实例都能获取到这个数组。除非你在某个实例中对其进行了重写，实事上是进行了覆盖。

```
function Thing() {
}
Thing.prototype.things = [];


var thing1 = new Thing();
var thing2 = new Thing();
thing1.things.push("foo");
console.log(thing2.things); //logs ["foo"]
```

通常上面的做法是不正确的（译注：改变thing1的同时也影响了thing2）。如果你想每个实例互不影响，那么请在函数里创建这些值，而不是在原型上。

```
function Thing() {
    this.things = [];
}


var thing1 = new Thing();
var thing2 = new Thing();
thing1.things.push("foo");
console.log(thing1.things); //logs ["foo"]
```

```
console.log(thing2.things); //logs []
```

多个函数可以形成原型链，这样this便会在原型链上逐步往上找直到找到你想引用的值。

```
function Thing1() {
}
Thing1.prototype.foo = "bar";

function Thing2() {
}
Thing2.prototype = new Thing1();


var thing = new Thing2();
console.log(thing.foo); //logs "bar"
```

很多人便是利用这个特性在JS中模拟经典的对象继承。注意原型链底层函数中对**this**的操作会覆盖上层的值。

```
function Thing1() {
}
Thing1.prototype.foo = "bar";

function Thing2() {
    this.foo = "foo";
}
Thing2.prototype = new Thing1();

function Thing3() {
}
Thing3.prototype = new Thing2();


var thing = new Thing3();
console.log(thing.foo); //logs "foo"
```

我习惯将赋值到原型上的函数称作方法。上面某些地方便使用了方法这样的字眼，比如logFoo方法。这些方法中的this同样具有在原型链上查找引用的魔力。通常将最初用来创建实例的函数称作构造函数。

原型链方法中的this是从实例中的this开始住上查找整个原型链的。也就是说，如果原型链中某个地方直接对this进行赋值覆盖了某个变量，那么我们拿到 的是覆盖后的值。

```
function Thing1() {
}
Thing1.prototype.foo = "bar";
Thing1.prototype.logFoo = function () {
    console.log(this.foo);
}

function Thing2() {
    this.foo = "foo";
}
Thing2.prototype = new Thing1();


var thing = new Thing2();
thing.logFoo(); //logs "foo";
```

在JavaScript中，函数可以嵌套函数，也就是你可以在函数里面继续定义函数。但内层函数是通过闭包获取外层函数里定义的变量值的，而不是直接继承this。

```
function Thing() {
}
```

```
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    var info = "attempting to log this.foo:";
    function doIt() {
        console.log(info, this.foo);
    }
    doIt();
}


var thing = new Thing();
thing.logFoo();  //logs "attempting to log this.foo: undefined"
```

上面示例中，doIt 函数中的this指代是全局作用域或者是undefined如果使用了"use strict";声明的话。对于很多新手来说，理解这点是非常头疼的。

还有更奇葩的。把实例的方法作为参数传递时，实例是不会跟着过去的。也就是说，此时方法中的this在调用时指向的是全局this或者是undefined在声明了"use strict"时。

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    console.log(this.foo);
}

function doIt(method) {
    method();
}


var thing = new Thing();
thing.logFoo(); //logs "bar"
doIt(thing.logFoo); //logs undefined
```

所以很多人习惯将this缓存起来，用个叫self或者其他什么的变量来保存，以将外层与内层的this区分开来。

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    var self = this;
    var info = "attempting to log this.foo:";
    function doIt() {
        console.log(info, self.foo);
    }
    doIt();
}


var thing = new Thing();
thing.logFoo();  //logs "attempting to log this.foo: bar"
```

但上面的方式不是万能的，在将方法做为参数传递时，就不起作用了。

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    var self = this;
    function doIt() {
        console.log(self.foo);
    }
    doIt();
}
```

```
function doItIndirectly(method) {
    method();
}


var thing = new Thing();
thing.logFoo(); //logs "bar"
doItIndirectly(thing.logFoo); //logs undefined
```

解决方法就是传递的时候使用bind方法显示指明上下文，bind方法是所有函数或方法都具有的。

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    console.log(this.foo);
}

function doIt(method) {
    method();
}


var thing = new Thing();
doIt(thing.logFoo.bind(thing)); //logs bar
```

同时也可以使用apply或call 来调用该方法或函数，让它在一个新的上下文中执行。

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    function doIt() {
        console.log(this.foo);
    }
    doIt.apply(this);
}

function doItIndirectly(method) {
    method();
}


var thing = new Thing();
doItIndirectly(thing.logFoo.bind(thing)); //logs bar
```

使用bind可以任意改变函数或方法的执行上下文，即使它没有被绑定到一个实例的原型上。

```
function Thing() {
}
Thing.prototype.foo = "bar";


function logFoo(aStr) {
    console.log(aStr, this.foo);
}


var thing = new Thing();
logFoo.bind(thing)("using bind"); //logs "using bind bar"
logFoo.apply(thing, ["using apply"]); //logs "using apply bar"
logFoo.call(thing, "using call"); //logs "using call bar"
logFoo("using nothing"); //logs "using nothing undefined"
```

避免在构造函数中返回作何东西，因为返回的东西可能覆盖本来该返回的实例。

```
function Thing() {
    return {};
}
Thing.prototype.foo = "bar";


Thing.prototype.logFoo = function () {
    console.log(this.foo);
}


var thing = new Thing();
thing.logFoo(); //Uncaught TypeError: undefined is not a function
```

但，如果你在构造函数里返回的是个原始值比如字符串或者数字什么的，上面的错误就不会发生了，返回语句将被忽略。所以最好别在一个将要通过new来调用的构造函数中返回作何东西，即使你是清醒的。如果你想实现工厂模式，那么请用一个函数来创建实例，并且不通过new来调用。当然这只是个人建议。

诚然，你也可以使用Object.create从而避免使用new。这样也能创建一个实例。

```
function Thing() {
}
Thing.prototype.foo = "bar";


Thing.prototype.logFoo = function () {
    console.log(this.foo);
}


var thing =  Object.create(Thing.prototype);
thing.logFoo(); //logs "bar"
```

这种方式不会调用该构造函数。

```
function Thing() {
    this.foo = "foo";
}
Thing.prototype.foo = "bar";


Thing.prototype.logFoo = function () {
    console.log(this.foo);
}


var thing =  Object.create(Thing.prototype);
thing.logFoo(); //logs "bar"
```

正因为Object.create没有调用构造函数，这在当你想实现一个继承时是非常有用的，随后你可能想要重写构造函数。

```
function Thing1() {
    this.foo = "foo";
}
Thing1.prototype.foo = "bar";

function Thing2() {
    this.logFoo(); //logs "bar"
    Thing1.apply(this);
    this.logFoo(); //logs "foo"
}
Thing2.prototype = Object.create(Thing1.prototype);
Thing2.prototype.logFoo = function () {
    console.log(this.foo);
}
```

```
var thing = new Thing2();
```

对象中的this

可以在对象的任何方法中使用this来访问该对象的属性。这与用new得到的实例是不一样的。

```
var obj = {
    foo: "bar",
    logFoo: function () {
        console.log(this.foo);
    }
};

obj.logFoo(); //logs "bar"
```

注意这里并没有使用new，也没有用Object.create，更没有函数的调用来创建对象。也可以将函数绑定到对象，就好像这个对象是一个实例一样。

```
var obj = {
    foo: "bar"
};

function logFoo() {
    console.log(this.foo);
}

logFoo.apply(obj); //logs "bar"
```

此时使用this没有向上查找原型链的复杂工序。通过this所拿到的只是该对象身上的属性而以。

```
var obj = {
    foo: "bar",
    deeper: {
        logFoo: function () {
            console.log(this.foo);
        }
    }
};

obj.deeper.logFoo(); //logs undefined
```

也可以不通过this，直接访问对象的属性。

```
var obj = {
    foo: "bar",
    deeper: {
        logFoo: function () {
            console.log(obj.foo);
        }
    }
};

obj.deeper.logFoo(); //logs "bar"
```

DOM 事件回调中的this

在DOM事件的处理函数中，this指代的是被绑定该事件的DOM元素。

```
function Listener() {
    document.getElementById("foo").addEventListener("click",
       this.handleClick);
}
Listener.prototype.handleClick = function (event) {
    console.log(this); //logs "<div id="foo"></div>"
}

var listener = new Listener();
document.getElementById("foo").click();
```

除非你通过bind人为改变了事件处理器的执行上下文。

```
function Listener() {
    document.getElementById("foo").addEventListener("click",
        this.handleClick.bind(this));
}
Listener.prototype.handleClick = function (event) {
    console.log(this); //logs Listener {handleClick: function}
}

var listener = new Listener();
document.getElementById("foo").click();
```

HTML中的this

HTML标签的属性中是可能写JS的，这种情况下this指代该HTML元素。

```
<div id="foo" onclick="console.log(this);"></div>
<script type="text/javascript">
document.getElementById("foo").click(); //logs <div id="foo"...
</script>
```

重写this

无法重写this，因为它是一个关键字。

```
function test () {
    var this = {};  // Uncaught SyntaxError: Unexpected token this
}
```

eval中的this

eval 中也可以正确获取当前的 this。

```
function Thing () {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    eval("console.log(this.foo)"); //logs "bar"
}

var thing = new Thing();
```

```
thing.logFoo();
```

这里存在安全隐患。最好的办法就是避免使用eval。

使用Function关键字创建的函数也可以获取this：

```
function Thing () {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = new Function("console.log(this.foo);");

var thing = new Thing();
thing.logFoo(); //logs "bar"
```

使用with时的this

使用with可以将this人为添加到当前执行环境中而不需要显示地引用this。

```
function Thing () {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    with (this) {
        console.log(foo);
        foo = "foo";
    }
}

var thing = new Thing();
thing.logFoo(); // logs "bar"
console.log(thing.foo); // logs "foo"
```

正如很多人认为的那样，使用with是不好的，因为会产生歧义。

jQuery中的this

一如HTML DOM元素的事件回调，jQuery库中大多地方的this也是指代的DOM元素。页面上的事件回调和一些便利的静态方法比如$.each 都是这样的。

```
<div class="foo bar1"></div>
<div class="foo bar2"></div>
<script type="text/javascript">
$(".foo").each(function () {
    console.log(this); //logs <div class="foo...
});
$(".foo").on("click", function () {
    console.log(this); //logs <div class="foo...
});
$(".foo").each(function () {
    this.click();
});
</script>
```

传递 this

如果你用过underscore.js或者lo-dash你便知道，这两个库中很多方法你可以传递一个参数来显示指定执行的上下文。比如

_.each。自ECMAScript 5 标准后，一些原生的JS方法也允许传递上下文，比如forEach。事实上，上文提到的bind，apply还有call 已经给我们手动指定函数执行上下文的能力了。

```
function Thing(type) {
    this.type = type;
}
Thing.prototype.log = function (thing) {
    console.log(this.type, thing);
}
Thing.prototype.logThings = function (arr) {
   arr.forEach(this.log, this); // logs "fruit apples..."
   _.each(arr, this.log, this); //logs "fruit apples..."
}

var thing = new Thing("fruit");
thing.logThings(["apples", "oranges", "strawberries", "bananas"]);
```

这样可以使得代码简洁些，不用层层嵌套bind，也不用不断地缓存this。

一些编程语言上手很简单，比如Go语言手册可以被快速读完。然后你差不多就掌握这门语言了，只是在实战时会有些小的问题或陷阱在等着你。

而JavaScript不是这样的。手册难读。非常多缺陷在里面，以至于人们抽离出了它好的部分（The Good Parts）。最好的文档可能是MDN上的了。所以我建议你看看他上面关于this的介绍，并且始终在搜索JS相关问题时加上"mdn" 来获得最好的文档资料。静态代码检查也是个不错的工具，比如jshint。

参考资料

- all this
- 详解this

在JavaScript中一切都是对象吗？

“在JavaScript中的一切都是对象”这个说法一直让我困惑。他们指的是什么？一个函数或者数组，它们怎么同时也是一个对象？在我们解答这个问题前，我们需要知道JavaScript是如何对不同数据类型归类的。

数据类型

在JavaScript中，有两个数据类型：基本类型和对象类型（对象类型有时候也被称为引用类型）。

基本类型

Number, String, Boolean, null, undefined

对象类型

Function, Object, Array

根据这个分类，这个问题的简单答案是：JavaScript中，并非一切值都是对象。是只有属于对象类型的值才是对象。也可以认为，任何非基本类型的都是对象类型。但是基本类型和对象类型有什么区别呢？更重要的是，人们所说的“所有”或“几乎所有”的JavaScript类似都是对象的真正含义是什么？这里说说主要的两个区别：可变性和比较。

Mutability可变性

根据我的经验，人们说的值是“类似对象”的真实含义是因为他们的可变性，更具体的说，是支持添加和删除属性。例如，因为函数和数组属于对象类型，你可以像对象一样给它们添加属性。

```
var func = function() {};
func.firstName = "Andrew";
func.firstName; // "Andrew"

var arry = [];
arry.age = 26;
arry.age; // 26
```

可变性开启了各种精彩的使用方式，也是理解原型和构造函数是如何工作的关键。

但是基本类型是不可变的，我们无法给它们添加属性。如下面代码所示，即使我们给基本类型添加了属性，解析器会无法在下一步读取它们的值。

```
var me = "Andrew";
me.lastname = "Robbins";
me.lastname; // undefined

var num = 10;
num.prop = 11;
num.prop; // undefined
```

在这一点上，基本类型的值无法改变的真正含义，需要在更基本的层面检查问题，如下代码所示:

```
1 = 2; // ReferenceError
```

这似乎是一个愚蠢的例子，但是我认为能反映出我们现在讨论的可变性，当你在JavaScript控制器中输入数字 1，编译器会给其分配基本类型，所以当你尝试将数字1改变成数字2会失败。

比较和传递

除了可变性，另外一个基本类型和对象类型重要的区别是他们在程序中比较和传递的方式。基本类型通过值来比较，而对象类型通过引用来比较，这是什么意思呢？我们先看看基本类型，如下代码所示：

```
"a" === "a"; // true
```

因为值“a”等于“a”所以为true，当我们在图中引入变量会发生什么呢？除了将一个基本类型储存在变量中什么也没发生。

```
var a = "a",
    b = "a";

a === b; // true
```

当基本类型通过值来比较，结果为true，变量a的值正好等于变量b的值，换句话说，”a”等于”a”。但是看看下面这个例子，如果我们在对象类型中应用相同的例子，我们会得到相反的结果。

```
var a = {name: "andrew"},
    b = {name: "andrew"};

a === b; // false
```

为什么会这样呢？如果想要两者比较为真需要对象类型要引用同样的类型。通过以上的例子，我们给变量b创建一个新的对象。就像David Flanagan说过：我们说的通过引用进行对象比较是:两个对象的值是否相同取决于它们是否指向相同的底层对象。

那我们这样传值会发生什么？

```
var a = {name: "andrew"},
    b = a;

b.name = "robbins";

a === b; // true
```

这个可能开始看上去很奇怪，但是仔细看看发生了什么，因为对象是对象类型的一部分，它比较的值是按引用进行传递。引用的是相同的底层对象。在以上的例子中，我们设置b等于a。并没有创建新对象，我们只是简单地创建了一个对其他对象的引用。从另一个方面来看我们是将变量b指向a，所以当我们改变b的name属性，我们同样改变了a的name属性。

如果将相同的例子应用在基本类型上呢？

```
var a = "Andrew",
    b = a;

b = "Robbins";

a === b; // false
```

当我们设置b等于a，请记住基本类型通过值来传递和比较，我们实际上另外创建了一个a的拷贝，所以我们改变b的值再跟a

比较，两个值是不一样的。

Wrapper Objects包装对象

有些人会说："好，如果基本类型不是对象，为什么我们可以调用他们的方法呢" 回答是包装对象。

当你尝试调用基本类型的方法，JavaScript在幕后做了一个巧妙的处理，将你的基本类型的值转换成临时对象用于构造函数，决定使用哪个构造函数取决于你尝试改变的基本类型的值，在String中调用.length会使用string()构造函数临时将基本类型转变成对象—允许你使用length方法而改变它，这个临时对象被称为包装对象。

有趣的是，null和undefined这两个基本类型不能调用这样的方法，否则会提示类型错误。

我们可以使用typeof来区分：

```
typeof "s"; // "string"
typeof new String(s); // "object"
```

备注：在执行typeof null 时js编译器会返回object，是显而易见的bug。

```
typeof null // "object"
```

当然，考虑到JavaScript是用10天写出来（http://www.quora.com/In-which-10-days-of-May-did-Brendan-Eich-write-JavaScript-Mocha-in-1995），就不过多去担忧了 :) 。

此外，我们也需要了解基本类型的属性是只读和临时的。

```
var hello = "hello";
hello.slice(1); // "ello" (Here we're actually calling slice not on hello, but of a copy of hello)
hello; // "hello"
```

Summary总结

JavaScript的值可以分为两种类型：基本类型和对象类型，基本类型有：String, Number, Boolean, Symbol, undefined 和 null.，对象类型有Function, Object 和 Array.

基本类型和对象类型的区别在于可变性和比较的方式以及程序中传值。

基本类型是不可变的，换种说法就是它们的值不能改变。对比而言，对象类型是可变的，它们的值可以更新和改变。

基本类型可以按值比较，当我们把一个基本类型赋值给另外一个基本类型，是复制了一个值。而对象这是通过引用进行比较，引用的是什么呢？引用的是底层对象。当我们赋值一个对象给另一个对象时。引用指针就创建了。在这个情况下，改变一个对象的值将更新另外一个对象的值。

当我们尝试在基本类型的值中调用方法时，JavaScript使用包装对象来临时控制基本类型，导致对象变为只读的并在垃圾回收后执行。

参考资料

- 在JavaScript中一切都是对象吗？

JavaScript进阶

讲解JS的高级用法，和一些工具。

underscore

这是一个js库文件，相当于一个工具包，提供了一些常用的功能扩展内置对象，比如Array, Function等，非常实用！

Underscore.js定义了一个下划线（_）对象，函数库的所有方法都属于这个对象。这些方法大致上可以分成：集合（collection）、数组（array）、函数（function）、对象（object）和工具（utility）五大类。

集合（collection）

这里提供的方法可供数组和对象使用。方便遍历，查找以及处理。

提供了类似ES5对Array增加的那些方法，比如 `map, each, some...`

数组函数（Array Functions）

提供的方法供数组使用，包含对数组的查找，分组，生成。

函数（Function (uh, ahem) Functions）

提供的方法用于函数上，包含对动态this的绑定,延时执行。

对象函数（Object Functions）

提供的方法供对象使用，包含检索,克隆等。以及一些isXXX的判断。

实用功能(Utility Functions)

产生随机数，html模板生成。

参考资料

- Underscore Home
- Underscore 中文API

Promise

Promise/Deferred 模型, 是一种异步编程的模式。其他的异步编程的模式，还有async来控制, 叫做流程控制。

现今流行的各大js库，几乎都不同程度的实现了Promise，如dojo，jQuery、Zepto、when.js、Q等，只是暴露出来的大都是Deferred对象。

Callback Hell

```
var fs = require('fs');
fs.readFile('sample01.txt', 'utf8', function (err, data) {
    fs.readFile('sample02.txt', 'utf8', function (err,data) {
        fs.readFile('sample03.txt', 'utf8', function (err, data) {
            fs.readFile('sample04.txt', 'utf8', function (err, data) {

            });
        });
    });
});
```

这样的嵌套就是令人憎恶的callback hell。

Promise/A+规范

- 一个promise可能有三种状态：等待（pending）、已完成（fulfilled）、已拒绝（rejected）
- 一个promise的状态只可能从“等待”转到“完成”态或者“拒绝”态，不能逆向转换，同时“完成”态和“拒绝”态不能相互转换
- promise必须实现then方法（可以说，then就是promise的核心），而且then必须返回一个promise，同一个promise的then可以调用多次，并且回调的执行顺序跟它们被定义时的顺序一致
- then方法接受两个参数，第一个参数是成功时的回调，在promise由“等待”态转换到“完成”态时调用，另一个是失败时的回调，在promise由“等待”态转换到“拒绝”态时调用。同时，then可以接受另一个promise传入，也接受一个“类then”的对象或方法，即thenable对象。

参考资料

- JavaScript Promise迷你书
- Promises/A+: 官方规范。
- Promises: 第三方个人整理的规范。
- html5rocks: JavaScript Promises
- JavaScript Promise启示录
- 《使用 promise 替代回调函数》
- Q
- When.js

callback问题

js编程遇到的最大问题就是单线程异步问题，这里面涉及最多的肯定就是callback了，不能处理好callback问题，常常会出现大量的嵌套情况，就是著名的`callback hell`了。

ES6中会引入一个新的规范，叫做Promise。这可以规范我们使用异步的情况。

releasing Zalgo

What it means is a function that accepts a callback and sometimes returns it right away, and some other times it returns it after some delay, in the future.

就是我们的代码之中的callback,可能sync，也可能async触发，比如：

```
function register(options, callback) {
    var first_name = (options['first_name'] || '').trim();
    var last_name = (options['last_name'] || '').trim();
    var errors = [];

    if (!first_name) {
        errors.push(['first_name', 'Please enter a valid name']);
    }
    if (!last_name) {
        errors.push(['last_name', 'Please enter a valid name']);
    }
    if (errors.length) {
        return callback(null, errors);
    }

    var params = {
        'user': {
            'email': options['email'],
            'first_name': first_name,
            'last_name': last_name,
            'new_password': options['new_password'],
            'new_password_confirmation': options['new_password_confirmation'],
            'terms': '1'
        },
        'vrid': options['vrid'],
        'merge_history': options['merge_history'] || 'true'
    };

    requestWithSignature('post', '/api/v2/users', params, callback);
}
```

而最好的做法，是保证callback全是sync或者async， 那么将上面的修改为:

```
if (errors.length) {
  process.nextTick(function() {
    callback(null, errors);
  });
  return;
}
```

就可以避免`releasing Zalgo`。

参考资料

- Callback 在大型编程时的一般性问题
- Designing APIs for Asynchrony
- Don't release Zalgo!

JavaScript设计模式

参考资料

- 深入理解JavaScript系列
- 常用的JavaScript设计模式
- 前端攻略系列(三) - javascript 设计模式（文章很长，请自备瓜子，水果和眼药水）
- JavaScript Patterns
- Learning JavaScript Design Patterns

从零开始写JavaScript框架

参考资料

- 从零开始编写JavaScript框架
- 我是怎么写JavaScript框架的（一）
- 专题：jQuery系列源码分析
- 慕课网:jQuery源码解析
- Vanilla JS

框架结构

第一个框架，我们选择模仿jQuery,这是目前web之中使用最广泛的一个库。

这个框架暂时就叫做mock.js, 模仿学习的意思。

闭包

js的变量作用域是按函数划分的，闭包是函数内部的函数，能够有效的保护好变量不污染外部。需要外部的接口，通过window挂载出去。比如这样:

```
(function() {
    var Mock = function() {

    };

    Mock.version = '0.0.1';

    window.M = Mock;
})();
```

模块

参考资料

- 逐行分析jQuery源码的奥秘
- 专题：jQuery系列源码分析

JavaScript模块管理

JavaScript模块管理

模块化简介

需要模块管理的原因就是JavaScript发展的越来越快，超过了它产生时候的自我定位。由于没有模块管理的概念，在做大型项目或者文件组织的时候，就会异常纠结。所以才会产生出这么多的模块管理工具。

CommonJS

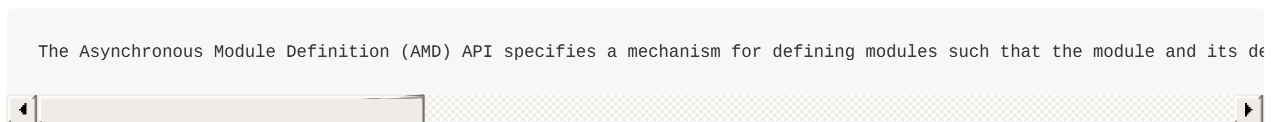
2009 ~ 2010 年间，CommonJS 社区大牛云集，稍微了解点历史的同学都清楚，在同时间出现了 nodejs，一下子让 javaScript 摇身一变，有了新的用武之地，同时在nodejs推动下的 CommonJS 模块系统也是逐渐深入人心:

- 通过 require 就可以引入一个 module，一个module通过 exports 来导出对外暴露的属性接口，在一个module里面没有通过 exports 暴露出来的变量都是相对于module私有的
- module 的查找也有一定的策略，通过统一的 package.json 来进行 module 的依赖关系配置，require一个module只需要 require package.json里面定义的name即可

这样的实现确实很好，但是一个最大的问题就是在浏览器加载脚本天生不支持同步的加载，无法通过文件I/O同步的require加载一个js脚本。So what ? CommonJS 中逐渐分裂出了 AMD，这个在浏览器环境有很好支持的module规范，其中最有代表性的实现则是 requirejs。

AMD

```
The Asynchronous Module Definition (AMD) API specifies a mechanism for defining modules such that the module and its de
```



翻译过来就是说：异步模块规范 API 定义了一种模块机制，这种机制下，模块和它的依赖可以异步的加载。这个非常适合于浏览器环境，因为同步的加载模块会对性能，可用性，debug调试，跨域访问产生问题。

确实，在浏览器环境下，AMD有着自己独特的优势： 由于源码和浏览器加载的一致，所见即所得，代码编写和debug非常方便。尤其是在多页面的web项目下，不同页面的脚本js都是根据依赖关系异步按需加载的，不用手动处理每个页面加载js脚本的情况。

但是，AMD 有一个不得不承认的作为一个module system的不足之处。请问？在 AMD(requireJS)里面怎么使用一个第三方库的？一般都会经历这么几个步骤：

- 使用的第三方库不想成为 global 的，只有引用的地方才可见
- 需要的库支不支持 AMD ？
- 不支持 AMD，我需要 fork 提个 patch 吗？
- 支持AMD，我的项目根路径在哪儿？库在哪儿？
- 不想要使用库的全部，要不要配置个 shim？
- 需不需要配置个 alias ？

一个库就需要问这么些个问题，而且都是人工手动的操作。最最关键的问题是你辛辛苦苦搞定的配置项都是相对于你当前项目的。当你想用在其他项目或者是单元测试，那么OK，你还得修改一下。因为，你相对的是当前项目的根路径，一旦根路径发生改变，一切都发生了变化。

requireJS 使用之前必须配置，同时该配置很难重用.

CommonJS in browser

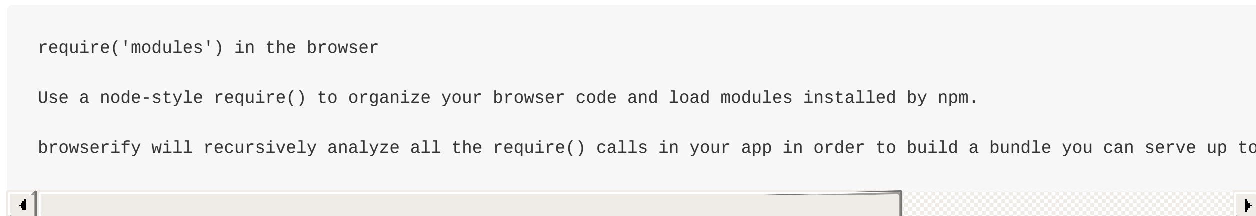
相比较于 CommonJS 里面如果要使用一个第三方库的话，仅仅只需要在 package.json 里面配置一下 库名和版本号，然后 npm install一下之后就可以直接 require 使用的方式，AMD 的处理简直弱爆了 !!!

对于 AMD 的这个不足之处，又有社区大神提出了可以在 browser 运行的 CommonJS 的方式，并且通过模块定义配置文件，可以很好的进行模块复用。比较知名的就有 substack 的 browserify, tj 曾主导的 component，还有后来的 duo，webpack，时代就转眼进入了 browser 上的 CommonJS.

由于 CommonJS 的 require 是同步的，在 require 处需要阻塞，这个在浏览器上并没有很好的支持（浏览器只能异步加载脚本，并没有同步的文件I/O），CommonJS 要在 browser 上直接使用则必须有一个 build 的过程，在这个 build 的过程里进行依赖关系的解析与做好映射。这里有一个典型的实现就是 substack 的 browserify。

browserify

browserify 在 github 上的 README.md 解释是：



```
require('modules') in the browser

Use a node-style require() to organize your browser code and load modules installed by npm.

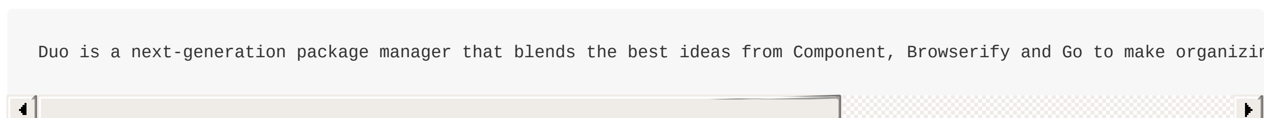
browserify will recursively analyze all the require() calls in your app in order to build a bundle you can serve up to
```

在 browserify 里可以编写 nodejs 一样的代码（即CommonJS以及使用package.json进行module管理），browserify会递归的解析依赖关系，并把这些依赖的文件全部build成一个bundle文件，在browser端使用则直接用 `<script>` tag 引入这个 bundle 文件即可.

component

component 通过 component.json 来进行依赖描述，它的库管理是基于 github repo的形式，由于进行了显示的配置依赖，它并不需要对源码进行 require 关系解析，但是时刻需要编写 component.json 也使得开发者非常的痛苦，开发者更希望 code over configuration 的形式

duo



```
Duo is a next-generation package manager that blends the best ideas from Component, Browserify and Go to make organizir
```

webpack

webpack 是一个 module bundler 即模块打包工具，它支持 CommonJS，AMD的module形式，同时还支持 code splittling，css 等

小结

这些 browser 上的 CommonJS 解决方案都有一个共同的问题，就是无法避免的需要一个 build 过程，这个过程虽然可以通过 watch task 来进行自动化，但是还是edit和debug还是非常不方便的

试想着，你在进行debug，你设置了一个debugger，然后单步调试，调试调试着跳到了另外一个文件中，然后由于是一个bundle大文件，你在浏览器开发者工具看到的永远都是同一个文件，然后你发现了问题所在，回头去改源码，还得先找到当前所在行与源码的对应关系！当然这个可以通过 source map 技术来进行解决，但是相比较 AMD 那种所见即所得的开发模式还是有一定差距

同时，需要build的过程也给多页面应用开发带来了很多麻烦，每个页面都要配置 watch task，都要配置 source map 之类

的，而且build过程如果一旦出现了build error，开发者还要去看看命令行里面的日志，除非使用 beefy 这种可以把命令行里面的日志输出到浏览器console，否则不知道情况的开发者就会一脸迷茫

CommonJS vs AMD

CommonJS

- 优点：简洁，更符合一个module system，同时 module 库的管理也非常方便
- 缺点：浏览器环境必须build才能使用，给开发过程带来不便

AMD

- 优点：天生异步，很好的与浏览器环境进行结合，开发过程所见即所得
- 缺点：不怎么简洁的module使用方式，第三方库的使用时的重复繁琐配置

更进一步的bearcat

bearcat 的一个理念可以用下面一句话来描述： Magic, self-described javaScript objects build up elastic, maintainable front-backend javaScript applications

bearcat 所倡导的就是使用简单、自描述的javaScript对象来构建弹性、可维护的前后端javaScript应用。当然可能有人会说，javaScript里面不仅仅是对象，还可以函数式、元编程什么的，其实也是要看应用场景的，bearcat更适合的场景是一个多人协作的、需要持续维护的系统（应用），如果是快速开发的脚本、工具、库，那么则该怎么简单、怎么方便，就怎么来。

参考资料

- Bearcat
- WebPack：更优秀的模块依赖管理工具，及require.js的缺陷
- browserify
- component
- duo
- webpack
- bearcat
- Javascript模块化编程（一）：模块的写法
- Javascript模块化编程（二）：AMD规范
- Javascript模块化编程（三）：require.js的用法

requireJS

参考资料

- RequireJS入门（一）
- RequireJS进阶（二）
- RequireJS进阶（三）

入门使用

还是先看官网: requirejs.org，中间就有一个getting start, 点击开始入门。

下载

要先使用，先要搭建环境，对于javascript文件，下载即可。进入GET REQUIREJS下载最新版本的requireJS文件。

在下载页面，还有其他几个文件:

- r.js: 优化，压缩文件的工具
- text: 加载文本文件的插件
- domReady: 确保文件在domReady后执行的插件
- cs: 支持CoffeeScript的插件
- i18n: 多语言版本支持的插件

添加到项目中

下面的例子假设所有的文件都在 `script` 目录下，目录结构为:

```
project-directory/
    project.html
    scripts/
        main.js
    helper/
        util.js
```

把requireJS加入到script目录下：

```
project-directory/
    project.html
    scripts/
        main.js
        require.js
    helper/
        util.js
```

接下来，在 `<head>` 中或者 `<body>` 之前，插入:

```
<!-- data-main attribute tells require.js to load
    scripts/main.js after require.js loads. -->
<script data-main="scripts/main" src="scripts/require.js"></script>
```

`data-main` 指出的脚本文件，是异步加载的。我们这里的 `main` 就是 `script/main.js` 文件，里面内容可以如下：

```
require(["helper/util"], function(util) {
    //This function is called when scripts/helper/util.js is loaded.
    //If util.js calls define(), then this function is not fired until
    //util's dependencies have loaded, and the util argument will hold
    //the module value for "helper/util".
});
```

这里提一下`require`和`define`的区别：

- require的依赖只要自身加载完成，就会调用
- define定义的，所有依赖都加载完了才会被触发

编写代码

真正详细的入门，其实在Require API。

但是一般而言，会在`data-mian`的文件中，写入配置文件， 类似:

```
require.config({
    /* global metadata */
    baseUrl: './',
    waitSeconds: 0,
    paths: {
        'domReady': 'bower_components/domReady/domReady',
        'modernizr': 'bower_components/modernizr/modernizr',
        "jquery": "bower_components/jquery/dist/jquery",
        'jquery.easing': 'bower_components/jquery.easing/js/jquery.easing'

    },
    shim: {
        'jquery.easing': ['jquery']
    },
    packages: [{
        name: 'gsap',
        main: '',
        location: 'bower_components/gsap/src/uncompressed'
    }]
});
```

压缩优化

开发时候的模式，各个模块其实都被加到了`head`部分，然后变成单独的请求，对于开发环境，我们有一个优化的重点，就是减少请求数。所以我们要使用工具，将开发环境的js文件进行合并压缩。

r.js

官方提供的一个优化工具，推荐自己再写一个build.js文件。比如:

```
({
    baseUrl: "./",
    name: "scripts/config",
    mainConfigFile: "scripts/config.js",
    out: "main-built.js",
    optimize: "none"
})
```

然后使用`r.js -o build.js`， 就可以进行文件的合并了。但这里是所有文件都在本地的情况。且压缩的是require依赖部分，实际请求数还是两个，另一个是require.js本身。如果需要合并为一个，再进行一次concat操作。

build.js使用的选项可以在all configuration options查询。

gulp

gulp-requirejs

项目地址:gulp-requirejs。目前处于out-of-date状态。

使用方法：

```
var gulpRequire = require('gulp-requirejs');

// optimize requireJS
gulp.task('optimize', function() {
    gulpRequire({
            baseUrl: "src",
            include: ['scripts/config'],
            mainConfigFile: "src/scripts/config.js",
            out: "main-built.js",
            optimize: "none"
        })
    .pipe(gulp.dest('dist/'));
});
```

gulp-requirejs-optimize

项目地址:gulp-requirejs-optimize。

grunt

grunt-contrib-requirejs

项目地址:grunt-contrib-requirejs

使用方法:

```
grunt.loadNpmTasks('grunt-contrib-requirejs');

requirejs: {
  compile: {
    options: {
      baseUrl: "path/to/base",
      mainConfigFile: "path/to/config.js",
      name: "path/to/almond", // assumes a production build using almond
      out: "path/to/optimized.js"
    }
  }
}
```

JavaScript数据结构

参考资料

- data-structure-with-js

数据类型

讲解数据结构之前，我想先理清一下JS中的基本数据类型。

JavaScript异步编程

常见的异步模式

高阶函数(泛函数)

```
step1(function(res1){
    step2(function(res2){
        step3(function(res3){
            //...
        });
    });
});
```

解耦程度特别低，如果送入的参数太多会显得很乱！这是最常见的一种方式，把函数作为参数送入，然后回调。

事件监听

```
f.on("evt", g);
function f(){
    setTimeout(function(){
        f.trigger("evt");
    })
}
```

JS 和 浏览器提供的原生方法基本都是基于事件触发机制的，耦合度很低，不过事件不能得到流程控制。

发布/订阅(Pub/Sub)

```
E.subscribe("evt", g);
function f(){
    setTimeout(function () {
       // f的任务代码
       E.publish("evt");
    }, 1000);
}
```

把事件全部交给 E 这个控制器管理，可以完全掌握事件被订阅的次数，以及订阅者的信息，管理起来特别方便。

Promise对象

Promise/A+ 规范是对 Promise/A 规范的补充和修改，他出现的目的是为了统一异步编程中的接口，JS中的异步编程是十分普遍的事情，也出现了很多的异步库，如果不统一接口，对开发者来说也是一件十分痛苦的事情。

在Promises/A规范中，每个任务都有三种状态：默认(pending)、完成(fulfilled)、失败(rejected)。

- 默认状态可以单向转移到完成状态，这个过程叫resolve，对应的方法是deferred.resolve(promiseOrValue)；
- 默认状态还可以单向转移到失败状态，这个过程叫reject，对应的方法是deferred.reject(reason)；
- 默认状态时，还可以通过deferred.notify(update)来宣告任务执行信息，如执行进度；
- 状态的转移是一次性的，一旦任务由初始的pending转为其他状态，就会进入到下一个任务的执行过程中。

参考资料

- JavaScript异步编程原理

什么是Promise

Promise/Deferred 模型, 是一种异步编程的模式。其他的异步编程的模式， 还有async来控制, 叫做流程控制。

现今流行的各大js库，几乎都不同程度的实现了Promise，如dojo，jQuery、Zepto、when.js、Q等，只是暴露出来的大都是Deferred对象。

Callback Hell

```
var fs = require('fs');
fs.readFile('sample01.txt', 'utf8', function (err, data) {
    fs.readFile('sample02.txt', 'utf8', function (err,data) {
        fs.readFile('sample03.txt', 'utf8', function (err, data) {
            fs.readFile('sample04.txt', 'utf8', function (err, data) {

            });
        });
    });
});
```

这样的嵌套就是令人憎恶的callback hell。

Promise/A+规范

- 一个promise可能有三种状态：等待（pending）、已完成（fulfilled）、已拒绝（rejected）
- 一个promise的状态只可能从“等待”转到“完成”态或者“拒绝”态，不能逆向转换，同时“完成”态和“拒绝”态不能相互转换
- promise必须实现then方法（可以说，then就是promise的核心），而且then必须返回一个promise，同一个promise的then可以调用多次，并且回调的执行顺序跟它们被定义时的顺序一致
- then方法接受两个参数，第一个参数是成功时的回调，在promise由“等待”态转换到“完成”态时调用，另一个是失败时的回调，在promise由“等待”态转换到“拒绝”态时调用。同时，then可以接受另一个promise传入，也接受一个“类then”的对象或方法，即thenable对象。

参考资料

- JavaScript Promise迷你书
- Promises/A+: 官方规范。
- Promises: 第三方个人整理的规范。
- html5rocks: JavaScript Promises
- JavaScript Promise启示录
- 《使用 promise 替代回调函数》
- Q
- When.js

Promise规范

从历史上说， Promises/A+ 规范将之前 Promises/A 规范的建议明确为了行为标准。其扩展了原规范以覆盖一些约定俗成的行为，以及省略掉一些仅在特定情况下存在的或者有问题的部分。

核心的 Promises/A+ 规范不设计如何建立、执行、拒绝 promises，而是专注于提供一个可互操作的 then 方法。上述对于 promises 的操作方法将来在其他规范中可能会触及。

Promise的状态

用 `new Promise` 实例化的promise对象有以下三个状态。

- **"has-resolution"** Fulfilled: resolve(成功)时。此时会调用 onFulfilled
- **"has-rejection"** Rejected: reject(失败)时。此时会调用 onRejected
- **"unresolved"** Pending: 既不是resolve也不是reject的状态。也就是promise对象刚被创建后的初始化状态等

关于上面这三种状态的读法，其中 左侧为在 ES6 Promises 规范中定义的术语， 而右侧则是在 Promises/A+ 中描述状态的术语。

promise对象的状态，从Pending转换为Fulfilled或Rejected之后， 这个promise对象的状态就不会再发生任何变化。也就是说，Promise与Event等不同，在.then 后执行的函数可以肯定地说只会被调用一次。

Then 方法

一个 promise 必须提供一个 then 方法以访问其当前值、最终返回值和据因。promise 的 then 方法接受两个参数：

```
promise.then(onFulfilled, onRejected)
```

1. onFulfilled 和 onRejected 都是可选参数。
    - 如果 onFulfilled 不是函数，其必须被忽略
    - 如果 onRejected 不是函数，其必须被忽略
2. 如果 onFulfilled 是函数：
    - 当 promise 执行结束后其必须被调用，其第一个参数为 promise 的值
    - 在 promise 执行结束前其不可被调用
    - 其调用次数不可超过一次
3. 如果 onRejected 是函数：
    - 当 promise 被拒绝执行后其必须被调用，其第一个参数为 promise 的据因
    - 在 promise 被拒绝执行前其不可被调用
    - 其调用次数不可超过一次
4. onFulfilled 和 onRejected 直到执行环境堆栈尽包含平台代码前不可被调用
5. onFulfilled 和 onRejected 必须被作为函数调用（即没有 this 值）
6. then 方法可以被同一个 promise 调用多次
    - 当 promise 成功执行时，所有 onFulfilled 需按照其注册顺序依次回调
    - 当 promise 被拒绝执行时，所有的 onRejected 需按照其注册顺序依次回调
7. then 方法必须返回一个 promise 对象, 如 `promise2 = promise1.then(onFulfilled, onRejected);`
    - 如果 onFulfilled 或者 onRejected 返回一个值 x ，则运行下面的 Promise 解决程序： `[[Resolve]](promise2, x)`
    - 如果 onFulfilled 或者 onRejected 抛出一个异常 e ，则 promise2 必须拒绝执行，并返回拒因 e
    - 如果 onFulfilled 不是函数且 promise1 成功执行， promise2 必须成功执行并返回相同的值
    - 如果 onRejected 不是函数且 promise1 拒绝执行， promise2 必须拒绝执行并返回相同的据因

Promise 解决程序

Promise解决程序是一个抽象的操作，其需输入一个 promose 和一个值，我们表示为 `[[Resolve]](promise, x)`，如果 x 是 thenable 的，同时若 x 至少满足和 promise 类似（即鸭子类型， x 拥有部分或全部 promise 拥有的方法属性）的前提，解决程序即尝试使 promise 接受 x 的状态；否则其用 x 的值来执行 promise 。

这种对 thenable 的操作允许 promise 实现互操作，只要其暴露出一个遵循 Promise/A+ 协议的 then 方法。

运行 `[[Resolve]](promise, x)` 需遵循以下步骤：

1. 如果 promise 和 x 指向同一对象，以 TypeError 为据因拒绝执行 promise
2. 如果 x 为 promise ，接受其状态
    - 如果 x 处于等待态， promise 需保持为等待态直至 x 被执行或拒绝
    - 如果 x 处于执行态，用相同的值执行 promise
    - 如果 x 处于拒绝态，用相同的据因拒绝 promise
3. 抑或 x 为对象或者函数
    - 设置 then 方法为 x.then
    - 如果 then 是函数，将 x 作为函数的作用域 this 调用之。其第一个参数为 resolvePromise ，第二个参数为 rejectPromise:
        - 如果 resolvePromise 以值 y 为参数被调用，则运行 `[[Resolve]](promise, y)`
        - 如果 rejectPromise 以据因 r 为参数被调用，则以据因 r 拒绝 promise
        - 如果 resolvePromise 和 rejectPromise 均被调用，或者被同一参数调用了多次，则优先采用首次调用和忽略剩下的调用
        - 如果调用 then 方法抛出了异常 e:
            - 如果 resolvePromise 或 rejectPromise 已经被调用，则忽略之
            - 否则以 e 为据因拒绝 promise
        - 如果 then 不是函数，以 x 为参数执行 promise
    - 如果 x 不为对象或者函数，以 x 为参数执行 promise

如果一个 promise 被一个循环的 thenable 链中的对象解决，而 `[[Resolve]](promise, thenable)` 的递归性质又使得其被再次调用，根据上述的算法将会陷入无限递归之中。算法不强制要求，但鼓励其实施者以检测这样的递归是否存在，若存在则以一个可识别的 TypeError 为据因来拒绝 promise。

参考资料

- Promises
- Promises/A+
- Promise/A+规范中文翻译
- MDN: Promise
- promise api 与应用场景
- JavaScript Promise迷你书（中文版）
- w3ctag/promises-guide: Promises指南 - 这里有很多关于概念方面的说明。
- domenic/promises-unwrapping: ES6 Promises规范的repo - 可以通过查看issue来了解各种关于规范的来龙去脉和信息。
- ECMAScript Language Specification ECMA-262 6th Edition – DRAFT: ES6 Promises的规范 - 如果想参考关于ES6 Promises的规范，则应该先看这里
- JavaScript Promises: There and back again - HTML5 Rocks: 关于Promises的文章 - 这里的示例代码和参考（reference）的完成度都很高

Promise实战

在很多需要与数据打交道的场合下，我们会遇到很多异步的情况,在异步操作的时候，我们还需要处理成功和失败两种情况，并且成功的时候，还可能需要把结果传递给下一个Ajax调用，从而形成"函数嵌套"的情况。callback是编写Javascript异步代码最最最简单的机制。可用这种原始的callback必须以牺牲控制流、异常处理和函数语义为代价。

具体实现的库

Polyfill

只需要在浏览器中加载Polyfill类库，就能使用IE10等或者还没有提供对Promise支持的浏览器中使用Promise里规定的方法。

- jakearchibald/es6-promise:一个兼容 ES6 Promises 的Polyfill类库。 它基于 RSVP.js 这个兼容 Promises/A+ 的类库，它只是 RSVP.js 的一个子集，只实现了Promises 规定的 API。
- yahoo/ypromise: 这是一个独立版本的 YUI 的 Promise Polyfill，具有和 ES6 Promises 的兼容性。
- getify/native-promise-only: 以作为ES6 Promises的polyfill为目的的类库 它严格按照ES6 Promises的规范设计，没有添加在规范中没有定义的功能。 如果运行环境有原生的Promise支持的话，则优先使用原生的Promise支持。

扩展类库

Promise扩展类库除了实现了Promise中定义的规范之外，还增加了自己独自定义的功能。

- then/promise: a super set of ES6 Promises designed to have readable, performant code and to provide just the extensions that are absolutely necessary for using promises today.
- petkaantonov/bluebird: 这个类库除了兼容 Promise 规范之外，还扩展了取消promise对象的运行，取得promise的运行进度，以及错误处理的扩展检测等非常丰富的功能，此外它在实现上还在性能问题下了很大的功夫。
- when: 大小很小,node和浏览器环境下都可以使用。
- q: 类库 Q 实现了 Promises 和 Deferreds 等规范。 它自2009年开始开发，还提供了面向Node.js的文件IO API Q-IO 等， 是一个在很多场景下都能用得到的类库。

编写Promise代码

创建流程

- 使用 `new Promise(fn)` 返回一个Promise对象
- 在 `fn` 中制定异步等处理:
  - 处理结果正常的情况，调用 `resolve(处理结果值)`
  - 处理结果错误的话,调用 `reject(Error对象)`

创建Promise对象

最基本的情况，是使用 `new Promise()` 来创建Promise对象。也可以使用 `Promise.resolve(value)` 代替 `new Promise()` 快捷方法。比如:

```
Promise.resolve(42);
// 等价于
new Promise(function(resolve) {
    reslove(42);
})
```

Thenable

就像我们有时称具有 `.length` 方法的非数组对象为Array like一样，thenable指的是一个具有 `.then` 方法的对象。

这种将thenable对象转换为promise对象的机制要求thenable对象所拥有的 then 方法应该和Promise所拥有的 then 方法具有同样的功能和处理过程，在将thenable对象转换为promise对象的时候，还会巧妙的利用thenable对象原来具有的 then 方法。变成了promise对象的话，就能直接使用 then 或者 catch, 比如:

```
var promise = Promise.resolve($.ajax('/json/comment.json'));// => promise对象
promise.then(function(value){
   console.log(value);
});
```

需要注意的是:即使一个对象具有 .then 方法，也不一定就能作为ES6 Promises对象使用。比如jQuery的 `Defeered Object` 的then方法机制与Promise不同。

其实在Promise里可以将任意个方法连在一起作为一个方法链（method chain），比如：

```
aPromise.then(function taskA(value){
// task A
}).then(function taskB(vaue){
// task B
}).catch(function onRejected(error){
    console.log(error);
});
```

Promise.reject

`Promise.reject(error)` 是和 `Promise.resolve(value)` 类似的静态方法，是 `new Promise()` 方法的快捷方式。

```
Promise.reject(new Error("出错了"));
// 等价于
new Promise(function(resolve,reject){
    reject(new Error("出错了"));
});
```

Promise的同步or异步调用

先看下面这段代码:

```
function onReady(fn) {
    var readyState = document.readyState;
    if (readyState === 'interactive' || readyState === 'complete') {
        fn();
    } else {
        window.addEventListener('DOMContentLoaded', fn);
    }
}
onReady(function () {
    console.log('DOM fully loaded and parsed');
});
console.log('==Starting==');
```

这段代码会根据执行时DOM是否已经装载完毕来决定是对回调函数进行同步调用还是异步调用。这实际上会让我们的代码是

同步还是一部产生混淆，所以为了解决这个问题，我们应该统一使用异步调用的方式:

```
function onReady(fn) {
    var readyState = document.readyState;
    if (readyState === 'interactive' || readyState === 'complete') {
        setTimeout(fn, 0);
    } else {
        window.addEventListener('DOMContentLoaded', fn);
    }
}
onReady(function () {
    console.log('DOM fully loaded and parsed');
});
console.log('==Starting==');
```

我们看到的 `promise.then` 也属于此类，为了避免上述中同时使用同步、异步调用可能引起的混乱问题，Promise在规范上规定 Promise只能使用异步调用方式，修改代码如下:

```
function onReadyPromise() {
    return new Promise(function (resolve, reject) {
        var readyState = document.readyState;
        if (readyState === 'interactive' || readyState === 'complete') {
            resolve();
        } else {
            window.addEventListener('DOMContentLoaded', resolve);
        }
    });
}
onReadyPromise().then(function () {
    console.log('DOM fully loaded and parsed');
});
console.log('==Starting==');
```

Promise#catch

链式上的catch会捕获前面所有then的错误情况。其实这也是个语法糖:

```
var promise = Promise.reject(new Error("message"));
promise.catch(function (error) {
    console.error(error);
});
// 等价于
var promise = Promise.reject(new Error("message"));
promise.then(undefined, function (error) {
    console.error(error);
});
```

提倡使用catch的原因还有一个就是: 使用promise.then(onFulfilled, onRejected) 的话, 在 onFulfilled 中发生异常的话，在 onRejected 中是捕获不到这个异常的。

然而实际上不管是 then 还是 catch 方法调用，都返回了一个新的promise对象。

Promise chain

通过 `then` 方法，我们可以将代码写成方法链的形式。比如:

```
function taskA() {
    console.log("Task A");
}
```

```
function taskB() {
    console.log("Task B");
}
function onRejected(error) {
    console.log("Catch Error: A or B", error);
}
function finalTask() {
    console.log("Final Task");
}

var promise = Promise.resolve();
promise
    .then(taskA)
    .then(taskB)
    .catch(onRejected)
    .then(finalTask);
```

chain的时候，如何传递参数？答案非常简单，那就是在 Task A 中 return 的返回值，会在 Task B 执行时传给它。因为return的值会由 Promise.resolve(return的返回值); 进行相应的包装处理。

多个Promise对象完成后统一处理

通过回调方式来进行多个异步调用

看代码:

```
function getURLCallback(URL, callback) {
    var req = new XMLHttpRequest();
    req.open('GET', URL, true);
    req.onload = function () {
        if (req.status === 200) {
            callback(null, req.responseText);
        } else {
            callback(new Error(req.statusText), req.response);
        }
    };
    req.onerror = function () {
        callback(new Error(req.statusText));
    };
    req.send();
}
// <1> 对JSON数据进行安全的解析
function jsonParse(callback, error, value) {
    if (error) {
        callback(error, value);
    } else {
        try {
            var result = JSON.parse(value);
            callback(null, result);
        } catch (e) {
            callback(e, value);
        }
    }
}
// <2> 发送XHR请求
var request = {
        comment: function getComment(callback) {
            return getURLCallback('http://azu.github.io/promises-book/json/comment.json', jsonParse.bind(null, callback
        },
        people: function getPeople(callback) {
            return getURLCallback('http://azu.github.io/promises-book/json/people.json', jsonParse.bind(null, callback)
        }
    };
// <3> 启动多个XHR请求，当所有请求返回时调用callback
function allRequest(requests, callback, results) {
    if (requests.length === 0) {
        return callback(null, results);
    }
    var req = requests.shift();
    req(function (error, value) {
```

```
        if (error) {
            callback(error, value);
        } else {
            results.push(value);
            allRequest(requests, callback, results);
        }
    });
}
function main(callback) {
    allRequest([request.comment, request.people], callback, []);
}
// 运行的例子
main(function(error, results){
    if(error){
        return console.error(error);
    }
    console.log(results);
});
```

缺点:

- 需要显示进行异常处理
- 为了不让嵌套层次太深，需要一个对request进行处理的函数
- 到处都是回调函数

使用**Promise#then**同时处理多个异步请求

```
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-book/json/comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-book/json/people.json').then(JSON.parse);
        }
    };
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    // [] 用来保存初始化的值
    var pushValue = recordValue.bind(null, []);
    return request.comment().then(pushValue).then(request.people).then(pushValue);
}
// 运行的例子
main().then(function (value) {
    console.log(value);
}).catch(function(error){
    console.error(error);
});
```

这种方法也不是我们期望的，和上面的回调函数风格相比:

- 可以直接使用 JSON.parse 函数
- 函数 main() 返回promise对象
- 错误处理的地方直接对返回的promise对象进行处理

Promise.all

`Promise.all` 接收一个 promise对象的数组作为参数，当这个数组里的所有promise对象全部变为resolve或reject状态的时候，它才会去调用 `.then` 方法。比如:

```
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-book/json/comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-book/json/people.json').then(JSON.parse);
        }
    };
function main() {
    return Promise.all([request.comment(), request.people()]);
}
// 运行示例
main().then(function (value) {
    console.log(value);
}).catch(function(error){
    console.log(error);
});
```

这样的优点是:

- main中的处理流程非常清晰
- Promise.all 接收 promise对象组成的数组作为参数

Promise数组是同时开始执行的，then调用参数的结果之中的results顺序和传递的数组的顺序一致。并且调用then的时间由最后一个完成的异步操作决定。

Promise.race

它的使用方法和Promise.all一样，接收一个promise对象数组为参数。与all的区别就是：**race**只要有一个**promise**对象进入 **FulFilled** 或者 **Rejected** 状态的话，就会继续进行后面的处理。但是Promise中的数组也还是会继续执行。但是then只接受第一个完成的Promise返回对象。

参考资料

- JavaScript异步编程的Promise模式

Async控制异步流程

Async是一个流程控制工具包，提供了直接而强大的异步功能。基于Javascript为Node.js设计，同时也可以直接在浏览器中使用。

参考资料

- Async
- Nodejs异步流程控制Async
- async_demo: 带中文注释的demo.

EventProxy控制异步流程

EventProxy是一个通过控制事件触发顺序来控制业务流程的工具。

参考资料

- eventproxy: An implementation of task/event based asynchronous pattern.

JSDeferred控制异步流程

参考资料

- JSDeferred

JavaScript正则表达式

正则表达式，乍一看以为是很高升的东西，但是对于程序员而言，它真的应该算是一个基础知识。我们在很多场合下都需要使用到它，这个技术又是一个比较通用的东西，所以对于程序员而言，是大有益处的。

参考资料

- 维基百科：正则表达式
- MDN: Regular Expressions
- stackoverflow: regular expression info
- Hacker News: Learn regular expressions in about 55 minutes
- 正则表达式30分钟入门
- Learn regular expressions in about 55 minutes
- regex one: 学习资料
- JavaScript Regular Expression Cheatsheet
- 在线验证工具，中文版
- Regex 101: 在线验证学习
- Regexper: 可视化正则表达式
- regex-tuesday: some challenges
- Regex Crossword: play to learn Regex

基本语法

既然作为前端，那就拿JavaScript语言作为例子，记录一下学习内容。

创建方式

- 字面值形式: `rtrim = /ab+c/g`
- new形式创建: `var replaceRegexp = RegExp("ab+c", 'g');`

这两种创建方式的区别是，如果你的正则表达式是可以确定的，那么使用字面值形式性能会更好，但是如果你的正则是动态创建的，那么只能选择构造函数的形式了。而在具体的使用上，二者是没有差别的。

书写规则

字面值形式的，记得表达式前后加上 `/` 符号。

- `\` : 转义字符，普通字符前面加上 `\` 可以代表特殊意义的字符。特殊意义的字符，也可以将其转换为普通字符。 当使用new RegExp("pattern")方法的时候不要忘记将\它自己进行转义，因为\在字符串里面也是一个转义字符。
- `^` : 匹配输入的开始。匹配行首的字符。
- `*` : 匹配前一个字符0次或者是多次。
- `+` : 匹配前面一个字符1次或者多次，和{1,}有相同的效果。
- `?` : 匹配前面一个字符0次或者1次，和{0,1}有相同的效果。
- `.` : （小数点）匹配任何除了新一行开头字符的任何单个字符。
- `(x)` : 匹配'x'并且记住匹配项。这个被叫做捕获括号。可以通过数组得到匹配的对象。
- `(?:x)` : 匹配'x'但是不记住匹配项。这种被叫做非捕获括号。
- `x(?=y)` : 匹配'x'仅仅当'x'后面跟着'y'.这种叫做正向肯定查找。
- `x(?!y)` : 匹配'x'仅仅当'x'后面不跟着'y',这个叫做正向否定查找。
- `x|y` : 匹配'x'或者'y'。
- `{n}` : n是一个正整数，匹配了前面一个字符刚好发生了n次。
- `{n,m}` : n 和 m 都是正整数。匹配前面的字符至少n次，最多m次。如果 n 或者 m 的值是0， 这个值被忽略。
- `[xyz]` : 一个字符集合。匹配方括号的中任意字符。你可以使用破折号（-）来指定一个字符范围。对于点（.）和星号（*）这样的特殊符号在一个字符集中没有特殊的意义。他们不必进行转意，不过转意也是起作用的。
- `[^xyz]` : 一个反向字符集。也就是说， 它匹配任何没有包含在方括号中的字符。你可以使用破折号（-）来指定一个字符范围。任何普通字符在这里都是起作用的。
- `[\b]` : 匹配一个退格(U+0008)。（不要和\b混淆了。）
- `\b` : 匹配一个词的边界。一个词的边界就是一个词不被另外一个词跟随的位置或者不是另一个词汇字符前边的位置。注意，一个匹配的词的边界并不包含在匹配的内容中。换句话说，一个匹配的词的边界的内容的长度是0。（不要和[\b]混淆了）
- `\B` : 匹配一个非单词边界。他匹配一个前后字符都是相同类型的位置：都是单词或者都不是单词。一个字符串的开始和结尾都被认为是非单词。
- `\cX` : 当X是处于A到Z之间的字符的时候，匹配字符串中的一个控制符。
- `\d` : 匹配一个数字。等价于[0-9]。
- `\D` : 匹配一个非数字字符。等价于[0-9]。
- `\f` : 匹配一个换页符 (U+000C)。
- `\n` : 匹配一个换行符 (U+000A)。
- `\r` : 匹配一个回车符 (U+000D)。
- `\s` : 匹配一个空白字符，包括空格、制表符、换页符和换行符。
- `\S` : 匹配一个非空白字符。
- `\t` : 匹配一个水平制表符 (U+0009)。
- `\v` : 匹配一个垂直制表符 (U+000B)。

- `\w`：匹配一个单字字符（字母、数字或者下划线）。等价于[A-Za-z0-9_]。
- `\W`：匹配一个非单字字符。等价于A-Za-z0-9_。
- `\n`：当 n 是一个正整数，一个返回引用到最后一个与有n插入的正值表达式(counting left parentheses)匹配的副字符串。
- `\0`：匹配 NULL (U+0000) 字符， 不要在这后面跟其它小数，因为 \0 是一个八进制转义序列。
- `\xhh`：匹配带有两位小数代码（hh）的字符。
- `\uhhhh`：匹配带有四位小数代码（hh）的字符。

如果需要记住匹配项，需要使用括号，这样才能通过数组查找到。

使用括号

`/Chapter (\d+)\.\d*/` 这部分中的 `(\d+)` 部分是会被记忆的，是可以作为一个字符串被使用。

可使用的方法

在RegExp对象的方法中有：

- exec: A RegExp method that executes a search for a match in a string. It returns an array of information.
- test: A RegExp method that tests for a match in a string. It returns true or false.

在String对象的方法中有:

- match:A String method that executes a search for a match in a string. It returns an array of information or null on a mismatch.
- replace:A String method that executes a search for a match in a string, and replaces the matched substring with a replacement substring.
- search: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match.
- split: A String method that uses a regular expression or a fixed string to break a string into an array of substrings.

exec

这是RegExp对象的一个方法，传入一个字符串，默认返回一个数组，包含匹配结果，以及匹配的一些信息。如果是带记忆信息的，数组的长度会变化。没有匹配的时候，返回null.

test

这是RegExp对象的一个方法，传入一个字符串，默认返回true或者false.

参考资料

- MDN: Regular Expressions
- stackoverflow: regular expression info
- 正则表达式30分钟入门
- Learn regular expressions in about 55 minutes
- regex one: 学习资料
- JavaScript Regular Expression Cheatsheet
- 在线验证工具，中文版
- Regex 101: 在线验证学习
- Regexper: 可视化正则表达式
- regex-tuesday: some challenges

- Regex Crossword: play to learn Regex

实用案例

参考资料

- 实战正则表达式

jQuery相关

jQuery事件注册和取消

jQuery毕竟是和DOM打交道最常用的库，它为我们提供了很好的兼容性保障。关于jQuery提供的事件相关方法，可以参考:event-handler-attachment。

事件注册

bind

这是1.x早起版本的一种绑定方式，你看名称就比较直接。主要适用于直接绑定到已经存在的对象上，但是在1.7版本之后，推荐使用的绑定方式是 `on` 方法，原因很简单，支持事件委托嘛(I guess)! 并且jQuery自身代码中，所有与事件注册相关的操作使用的都是on方法。

还有一点需要提到的，就是通过bind绑定的事件，可以通过unbind取消。

delegate

用作事件代理代理。

live

也是用作事件代理，区别就是绑定的选择器每次都会更新。已经弃用了！

on

上面的三个方法: `bind, delegate, live` 就是早期jQuery事件处理的主要方法，但是 `on` 出现之后，都被替换掉了。也就是说，凡事和事件打交道的时候，都要使用 `on` ，这才是老大！

one

用法和on类似，触发一次后自动取消。

off

取消事件绑定。

trigger

手动触发事件。

命名空间

使用了很久jQuery的事件绑定，才在同事那里学到可以添加命名空间这个概念。充分表明了文档的重要性，因为这个是文档里提及到的，并且就在前几段！！！

首先jQuery绑定的事件名称可以是任意String，因为可以通过 `trigger` 触发，对于不同的事件名称，可以通过添加命名空间的形式进行区分，比如 `click` 事件，我们可以添加 `click.namespace` ，也可以是 `click.name.space` ，多个命名空间并行。其实主要原因就是jQuery将每一个绑定事件对应的处理函数都存储起来了，添加命名空间，也就是对相同事件上的不同或多处事件处理函数进行区分。

这样在 `trigger` 的时候，就可以进行区分了，从而不触发其他绑定的处理函数。

事件代理

jQuery中事件流是冒泡的，也就是从触发的那个元素(event.target)开始，向document冒泡。我们可以通过 `event.stopPorpagation()` 停止冒泡。

事件代理的一大好处就是可以绑定一个元素上，检测子元素触发，而不需要绑定在N多个相同的子元素上，这样效率上会好很多。

事件对象

Event Object，jQuery的事件对象是遵守W3C标准的。关于事件对象的详细内容，参见: Event Object。

传递**data**

jquery的绑定事件，一般都会有一个 `[data]` 参数，可以传递字符串orjson对象，然后通过事件对象 `event.data` 获取。

性能问题

事件影响我们程序效率的情况，不是那些 `click` 这样触发不频繁的事件，而是 `mousemove, scroll` 这样一分钟内会触发多次的事件，我们要在这些处理函数中，尽量减小工作量,缓存需要查询的元素,并且尽量限制触发频率。（通过setTimeout)

影响性能的另外一大原因就是事件代理了，虽然它好用，但是如果我们把所有事件都绑定在document上，那么每次都是经历一个大的遍历过程，这是消耗性能的一个问题。解决的办法就是尽量将几个可以使用事件代理的处理，找一个最近的对象进行绑定。

事件代理的选择器也是一个讲究，使用 `"#myForm", "a.external", "button"` 这类会比较快，这和css的选择器道理类似。

获取元素在DOM中的顺序

jQuery事件中的this和标准事件处理中的this是一致的，都是指向绑定事件的DOM元素。

事件中的元素

- 事件中的this指针,指向的都是绑定事件的元素。
- event.currentTarget, 和this一致
- event.delegateTarget: 如果使用了事件委托，那就是对于的绑定元素。否则和currentTarget一致
- event.target: 真正触发的元素。

搜索元素

eq()方法

这个主要是过滤选取的元素，比如 `$('li').eq(n)` 。

index()方法

索引值是从0记数的，返回元素在其同级别DOM元素中的位置。

- 如果不传递参数，返回的是当前元素的索引值。
- 如果传递一个DOM元素进去，返回的是在当前选择器中的索引。

get()方法

传入索引值，获取对于的DOM元素。

遍历元素

parent()方法

- 不传递参数，返回直接的父级元素。
- 传入选择器， 在返回的父级元素中过滤符合的元素，返回数组，可以通过判断数组长度判断是否存在指定父级元素。

parents()方法

和parent方法类似，不过返回的不是直接父级元素，而是所有的祖先元素。还有一点不同的是, parent遍历的顶点是document:

```
$( "html" ).parent(); // [document]
$( "html" ).parents(); // []
```

parentsUntil()方法

Deferred对象

jQuery提供一个静态方法 `Deferred`，返回一个可供链式调回调函数队列。为的是解决异步编程问题，遵循Common Promise/A规范。jQuery在1.5版本中引入了这个特性，并为其重写了Ajax模块，可见其重要性。

它解决了如何处理耗时操作的问题，对那些操作提供了更好的控制，以及统一的编程接口。

deferred对象的方法

1. $.Deferred() 生成一个deferred对象
2. deferred.done() 指定操作成功时的回调函数
3. deferred.fail() 指定操作失败时的回调函数
4. deferred.promise() 没有参数时，返回一个新的deferred对象，该对象的运行状态无法被改变；接受参数时，作用为在参数对象上部署deferred接口。
5. deferred.resolve() 手动改变deferred对象的运行状态为"已完成"，从而立即触发done()方法。
6. deferred.reject() 这个方法与deferred.resolve()正好相反，调用后将deferred对象的运行状态变为"已失败"，从而立即触发fail()方法。
7. $.when() 为多个操作指定回调函数
8. deferred.then(): 把done和fail结合写在一起了，传递两个回调函数即可。
9. deferred.always(): 这个方法也是用来指定回调函数的，它的作用是，不管调用的是deferred.resolve()还是deferred.reject()，最后总是执行。

参考资料

- jQuery.Deferred
- jQuery的deferred对象详解
- 读jQuery之二十（Deferred对象）

jQuery代码技巧

一些优化性能的技巧

- 缓存变量
- 避免全局变量
- 使用匈牙利命名法
- 使用 Var 链（单 Var 模式）
- 请使用On方法注册事件
- 合并同一个对象上的多次操作，比如修改样式
- 链式操作
- 维持代码的可读性
- 选择短路求值, `&&` 和 `||`
- 选择捷径
- 繁重的操作中分离元素: 如果你打算对DOM元素做大量操作（连续设置多个属性或css样式），建议首先分离元素然后在添加。
- 熟记技巧, 比如: `$.data('#id', key, value)` 比 `$('#id').data(key, value)` 高效
- 使用子查询缓存的父元素: DOM遍历是一项昂贵的操作。典型做法是缓存父元素并在选择子元素时重用这些缓存元素。
- 避免通用选择符: 将通用选择符放到后代选择符中，性能非常糟糕。
- 避免隐式通用选择符: 通用选择符有时是隐式的，不容易发现。
- 优化选择符: Id选择符应该是唯一的，所以没有必要添加额外的选择符。
-

参考资料

- 编写更好的jQuery代码的建议
- 50个jQuery代码段帮你成为更好的JavaScript开发者

jQuery源码分析

参考资料

- 逐行分析jQuery源码的奥秘
- 专题：jQuery系列源码分析
- 慕课网:jQuery源码解析

理解架构

Write less, do more.

代码结构

```
(function( global, factory ) {

    if ( typeof module === "object" && typeof module.exports === "object" ) {
        // For CommonJS and CommonJS-like environments where a proper `window`
        // is present, execute the factory and get jQuery.
        // For environments that do not have a `window` with a `document`
        // (such as Node.js), expose a factory as module.exports.
        // This accentuates the need for the creation of a real `window`.
        // e.g. var jQuery = require("jquery")(window);
        // See ticket #14549 for more info.
        module.exports = global.document ?
            factory( global, true ) :
            function( w ) {
                if ( !w.document ) {
                    throw new Error( "jQuery requires a window with a document" );
                }
                return factory( w );
            };
    } else {
        factory( global );
    }

// Pass this if window is not defined yet
}(typeof window !== "undefined" ? window : this, function( window, noGlobal ) {
var jQuery = function( selector, context ) {
        return new jQuery.fn.init( selector, context );
    };
    jQuery.fn = jQuery.prototype = {};
    // 核心方法
    // 回调系统
    // 异步队列
    // 数据缓存
    // 队列操作
    // 选择器引
    // 属性操作
    // 节点遍历
    // 文档处理
    // 样式操作
    // 属性操作
    // 事件体系
    // AJAX交互
    // 动画引擎
    return jQuery;
}));
```

模块依赖

jQuery一共13个模块，从2.1版开始jQuery支持通过AMD模块划分，jQuery在最开始发布的1.0版本是很简单的，只有CSS选择符、事件处理和AJAX交互3大块。

jQuery也可理解分为五大块，选择器、DOM操作、事件、AJAX与动画，那么为什么有13个模块？因为jQuery的设计中最喜欢的做的一件事，就是抽出共同的特性使之“模块化”，当然也是更贴近S.O.L.I.D五大原则的“单一职责SRP”了，遵守单一职责的好处是可以让我们很容易地来维护这个对象，比如，当一个对象封装了很多职责的时候，一旦一个职责需要修改，势必会影响该对象的其它职责代码。通过解耦可以让每个职责更加有弹性地变化。

模块列表如图：

立即调用表达式

任何库与框架设计的第一个要点就是解决命名空间与变量污染的问题。jQuery就是利用了JavaScript函数作用域的特性，采用立即调用表达式包裹了自身的方法来解决这个问题。

比如：

```
(function(window, undefined) {
    var jQuery = function() {}
    // ...
    window.jQuery = window.$ = jQuery;
})(window);
```

从上面的代码可看出，自动初始化这个函数，让其只构建一次。这种写法的优势：

- window和undefined都是为了减少变量查找所经过的scope作用域。当window通过传递给闭包内部之后，在闭包内部使用它的时候，可以把它当成一个局部变量，显然比原先在window scope下查找的时候要快一些。
- undefined也是同样的道理，其实这个undefined并不是JavaScript数据类型的undefined，而是一个普普通通的变量名。只是因为没给它传递值，它的值就是undefined，undefined并不是JavaScript的保留字。

Q:为什么传递undefined?

Javascript 中的 undefined 并不是作为关键字，因此可以允许用户对其赋值。

大部分浏览器都是不能被修改的,但是IE8存在这个问题, 比如：

```
var undefined = 'xxx'
;(function(window) {
  alert(undefined);//IE8 'xxx'
})(window)
```

jQuery的类数组对象结构

很多人迷惑的jQuery为什么能像数组一样操作，通过对象get方法或者直接通过下标0索引就能转成DOM对象。

比如jQuery的入口都是统一的$, 通过传递参数的不同，实现了9种方法的重载：

```
jQuery([selector,[context]])
jQuery(element)
jQuery(elementArray)
jQuery(object)
jQuery(jQuery object)
jQuery(html,[ownerDocument])
jQuery(html,[attributes])
jQuery()
jQuery(callback)
```

9种用法整体来说可以分三大块：选择器、dom的处理、dom加载。运用了设计模式里面的 `工厂模式` 。

所以为了更方便这些操作，让节点与实例对象通过一个桥梁给关联起来，jQuery内部就采用了一种叫“类数组对象”的方式作为存储结构，所以我们即可以像对象一样处理jQuery操作，也能像数组一样可以使用push、pop、shift、unshift、sort、each、map等类数组的方法操作jQuery对象了。

抽象的表示创建可用数组下标操作的对象:

```
var aQuery = function(selector) {
    //强制为对象
    if (!(this instanceof aQuery)) {
        return new aQuery(selector);
    }
    var elem = document.getElementById(/[^#].*/.exec(selector)[0]);
    this.length = 1;
    this[0] = elem;
    this.context = document;
    this.selector = selector;
    this.get = function(num) {
        return this[num];
    }
    return this;
}
```

jQuery的无new构建原理

函数aQuery()内部首先保证了必须是通过new操作符构建。这样就能保证当前构建的是一个带有this的实例对象，既然是对象我们可以把所有的属性与方法作为对象的key与value的方式给映射到this上，所以如上结构就可以模拟出jQuery的这样的操作了，即可通过索引取值，也可以链式方法取值，但是这样的结构是有很大的缺陷的，每次调用ajQuery方法等于是创建了一个新的实例，那么类似get方法就要在每一个实例上重新创建一遍，性能就大打折扣，所以jQuery在结构上的优化不仅仅只是我们看到的，除了实现类数组结构、方法的原型共享，而且还实现方法的静态与实例的共存，这是我们之后将会重点分析的。

jQuery中ready与load事件

jQuery中存在三种文档加载的方法：

```
$(document).ready(function() {
    // ...代码...
})
//document ready 简写
$(function() {
    // ...代码...
})
$(document).load(function() {
    // ...代码...
})
```

DOM文档加载的步骤

- 解析HTML结构。
- 加载外部脚本和样式表文件。
- 解析并执行脚本代码。
- 构造HTML DOM模型。//ready
- 加载图片等外部文件。
- 页面加载完毕。//load

jQuery多库共存处理

多库共存换句话说可以叫无冲突处理。jQuery给出了解决方案： `noConflict函数` 。使用demo:

```
jQuery.noConflict();
// 使用 jQuery
jQuery("aaron").show();
// 使用其他库的 $()
$("aaron").style.display = ‘block’;
```

这个函数必须在你导入jQuery文件之后，并且在导入另一个导致冲突的库之前使用。当然也应当在其他冲突的库被使用之前，除非jQuery是最后一个导入的。

参考资料

- jQuery 2.0.3 源码分析core - 整体架构

GSAP

常见问题

TweenMax当seek的时候，不执行onComplete

描述：在使用TimelineMax或者TweenMax添加过渡的时候，如果添加的内容包含了`onComplete`，seek的时候是不会执行的。因为不在最外层的`onComplete`。

跨域问题

跨域问题

同源策略

如果两个页面拥有相同的协议（protocol），端口（如果指定），和主机，那么这两个页面就属于同一个源（origin）。

在以前，前端和后端混杂在一起， 比如JavaScript直接调用同系统里面的一个Httphandler，就不存在跨域的问题，但是随着现代的这种多种客户端的流行，比如一个应用通常会有Web端，App端，以及WebApp端，各种客户端通常会使用同一套的后台处理逻辑，即API， 前后端分离的开发策略流行起来，前端只关注展现，通常使用JavaScript，后端处理逻辑和数据通常使用WebService来提供json数据。一般的前端页面和后端的WebService API通常部署在不同的服务器或者域名上。这样，通过ajax请求WebService的时候，就会出现同源策略的问题。

解决方案

- JSONP (JSON with padding)
- CORS (Cross-origin resource sharing)

参考资料

- JavaScript 的同源策略

iframe自适应

WebService解决方案

参考资料

- 浅谈跨域以WebService对跨域的支持

JSONP

HTML 的 `<script>` 元素是一个例外。利用 `<script>` 元素的这个开放策略，网页可以得到从其他来源动态产生的 JSON 资料，而这种使用模式就是所谓的 JSONP。用 JSONP 抓到的资料并不是 JSON，而是任意的 JavaScript，用 JavaScript 直译器执行而不是用 JSON 解析器解析。

问题现象

1、一个众所周知的问题，Ajax直接请求普通文件存在跨域无权限访问的问题，甭管你是静态页面、动态网页、web服务、WCF，只要是跨域请求，一律不准；

2、不过我们又发现，Web页面上调用js文件时则不受是否跨域的影响（不仅如此，我们还发现凡是拥有"src"这个属性的标签都拥有跨域的能力，比如 `<script>`、 `<img>`、 `<iframe>`）；

3、于是可以判断，当前阶段如果想通过纯web端（ActiveX控件、服务端代理、属于未来的HTML5之Websocket等方式不算）跨域访问数据就只有一种可能，那就是在远程服务器上设法把数据装进js格式的文件里，供客户端调用和进一步处理；

4、恰巧我们已经知道有一种叫做JSON的纯字符数据格式可以简洁的描述复杂数据，更妙的是JSON还被js原生支持，所以在客户端几乎可以随心所欲的处理这种格式的数据；

5、这样子解决方案就呼之欲出了，web客户端通过与调用脚本一模一样的方式，来调用跨域服务器上动态生成的js格式文件（一般以JSON为后缀），显而易见，服务器之所以要动态生成JSON文件，目的就在于把客户端需要的数据装入进去。

6、客户端在对JSON文件调用成功之后，也就获得了自己所需的数据，剩下的就是按照自己需求进行处理和展现了，这种获取远程数据的方式看起来非常像AJAX，但其实并不一样。

7、为了便于客户端使用数据，逐渐形成了一种非正式传输协议，人们把它称作JSONP，该协议的一个要点就是允许用户传递一个callback参数给服务端，然后服务端返回数据时会将这个callback参数作为函数名来包裹住JSON数据，这样客户端就可以随意定制自己的函数来自动处理返回数据了。

原理分析

同源策略下，某个服务器是无法获取到服务器以外的数据，但是html里面的img,iframe和script等标签是个例外，这些标签可以通过src属性请求到其他服务器上的数据。而JSONP就是通过script节点src调用跨域的请求。

当我们向服务器提交一个JSONP的请求时,我们给服务传了一个特殊的参数,告诉服务端要对结果特殊处理一下。这样服务端返回的数据就会进行一点包装，客户端就可以处理。

举个例子，服务端和客户端约定要传一个名为callback的参数来使用JSONP功能。比如请求的参数如下:

```
http://www.example.net/sample.aspx?callback=mycallback
```

如果没有后面的callback参数，即不使用JSONP的模式，该服务的返回结果可能是一个单纯的json字符串，比如：`{ foo : 'bar' }`。但是如果使用JSONP模式，那么返回的是一个函数调用: `mycallback({ foo : 'bar' })`，这样我们在代码之中，定义一个名为mycallback的回调函数，就可以解决跨域问题了。

参考资料

- 维基百科: JSONP

测试相关

Blackbox

Blackbox允许屏蔽指定的JS文件，这样调试的时候就会绕过它们了。

屏蔽文件后会怎么样

- 库代码（被屏蔽的文件）里抛出异常时不会暂停（当设置为Pause on exceptions时）
- 调试时Stepping into/out/over都会忽略库代码
- 事件断点也会忽略库代码
- 库代码里设置的任何断点也不会起作用
- 最终的结果就是只会调试应用代码而忽略第三方代码（配置了Blackbox的代码）。

怎样屏蔽文件

开发人员工具的**Settings**面板

打开开发人员工具的配置面板，在Sources下点击Manage framework blackboxing, 有如下集中方式配置：

- 输入文件名称
- 用正则表达式匹配
  - 包含特定名称的文件，比如/backbone.js$
  - 特定类型的文件，比如.min.js$
- 输入整个文件夹，比如bower_components

另外，需要暂时不屏蔽某个规则时，可以将Behavior改为Disable。或者也可以直接删除（光标移到某行规则后会有个X）。Blackbox content scripts是指屏蔽Chrome插件注入页面的脚本.

在**Sources**面板上右键某个文件

在Sources面板目录里，或者编辑器里，右键点击“Blackbox Script”，可以将屏蔽该文件，同时也会增加到Setting面板中的匹配规则里。

参考资料

- 调试时屏蔽JavaScript库代码 –Chrome DevTools Blackbox功能介绍

Mocha测试框架

参考资料

- Mocha
- chai

前端自动化

自动化流程

前端的开发不同于后端，涉及的东西有很多，后台通常的流程就是:创建->编码->测试。而前端的制作，包含了切图->项目构建->编译(less,coffeescript)->兼容性->测试->发布等等多道流程。

常规的前端流程

- 搭建基础的项目骨架。包含创建模版（html、jade、haml）、脚本（javascript、coffeescript）、样式（css、less、sass、stylus）文件
- 启动本地服务器，比如MAMP
- 切图+编码页面，搭建界面
- 保存，刷新浏览器查看预览，并且进行调整
- 如果偶遇需要编译的文件，还要使用工具对其（jade、coffeescript、less、sass...）编译
- 编码交互和事件
- 保存，刷新浏览器，手动触发检查
- 执行测试用例
- 代码检测, 优化
- 移除调试代码
- 静态资源合并与优化, javascript和css文件的合并与压缩
- 打包上传到服务器
- 部署测试环境
- 灰度发布现网

缺点：在开发的中后期，代码会庞大，维护起来比较费力，修改和新增都担心对之前的会有影响，且还担心线上的版本有问题，需要重新部署。

工具化

以上的流程中，其实有些内容，比如切图，刷新，测试，部署都可以通过工具化的方式进行优化。

大平台公司对工具化的坚持是一致的：凡是被不断重复的过程，将其工具化，绑定到自动化流程之中。技术产品也需要Don't make me think的方式来推广最佳实践。总而言之：依靠工具，而不是经验。

nodejs的出现，不敢说改变后端的开发很多，但是对前端的工具化确实进步了很多。

自动化

有了工具之后，就是要想办法利用工具达到流程的自动化。

Yeoman

Yeoman是一款现代Web应用的脚手架工具，也是一个工作流。它包含了yo, grunt, bower三个工具。

yo

脚手架工具，帮助构建项目骨架。

bower

Bower是用于Web前端开发的包管理器。它运行在Git之上（因此必须先安装Git），默认情况下会去Github下载，并存放在 `bower_components` 目录下。

grunt

自动化工具，通过 `package.json` 来安装依赖的差距，使用 `gruntfile.js` 文件来编写要执行的任务。

gulp

grunt使用多了就会发现任务编写过程的麻烦，后来居上的gulp是一个很好的替代，我已经在使用，并且离不开了！

参考资料

- 停不下来的前端，自动化流程
- 前端自动化工作流简介
- 前端自动化构建和发布系统的设计（一）
- 前端自动化构建和发布系统的设计（二）

yeoman

Yeoman helps you to kickstart new projects, prescribing best practices and tools to help you stay productive.

如何开始

学习一个新东西，最直接的方式就是去官方网站找向导，比如Getting Start.

创建自己的generator

官方指导,主要是对模板文件创建的过程进行了解。详细的使用查看: API

index.js结构解析

```
'use strict';
// 引入使用的模块，yosay是命令行里显示文字的插件，chalk是增强命令行下颜色显示。
var yeoman = require('yeoman-generator');
var yosay = require('yosay');
var chalk = require('chalk');

// 导出模块，使得yo xxx能够运行
module.exports = yeoman.generators.Base.extend({
    // 默认会添加的构造函数
    constructor: function () {
        yeoman.generators.Base.apply(this, arguments);
    },
    // 初始化执行的内容，一般读取配置文件
    initializing: function () {
        this.pkg = require('../package.json');
    },
    // 提示信息相关内容，比如询问用户是否使用某些模块
    prompting: function () {
    },

    // 拷贝文件，创建真正的项目，这里面提三个需要注意的函数
    // template： 拷贝文件，同时会替换里面的配置信息
    // copy: 只负责拷贝，但是好像也能替换里面的配置信息
    // write: 修改文件内容
    writing: {

    }
})
```

bower

bower是twitter推出的一套前端组件管理工具。一般用来安装前端所需模块，npm安装的一般是node环境下的组件，或者开发的辅助工具。

.bowerrc

这个文件可以指定bower安装的位置，默认是:

```
{
    "directory": "src/bower_components"
}
```

一般会修改为:

```
{
    "directory": "src/scripts/vendor"
}
```

bower命令

如果存在 `package.json` 文件，那么 `bower install` 就会按照这个文件的依赖进行安装，如果自行安装的话，最好加上 `--saveDev` ，会自动添加到 `package.json` 文件中。

Gulp

前端自动化流程管理。在JavaScript的世界里，Grunt.js是基于Node.js的自动化任务运行器。2013年02月18日，Grunt v0.4.0 发布。Fractal公司积极参与了数个流行Node.js模块的开发，它去年发布了一个新的构建系统Gulp，希望能够取其精华，并取代Grunt，成为最流行的JavaScript任务运行器。

Gulp和Grunt的异同点

- 易于使用：采用代码优于配置策略，Gulp让简单的事情继续简单，复杂的任务变得可管理。
- 高效：通过利用Node.js强大的流，不需要往磁盘写中间文件，可以更快地完成构建。
- 高质量：Gulp严格的插件指导方针，确保插件简单并且按你期望的方式工作。
- 易于学习：通过把API降到最少，你能在很短的时间内学会Gulp。构建工作就像你设想的一样：是一系列流管道。

Gulp特点

- 易用：Gulp相比Grunt更简洁，而且遵循代码优于配置策略，维护Gulp更像是写代码。
- 高效：Gulp相比Grunt更有设计感，核心设计基于Unix流的概念，通过管道连接，不需要写中间文件。
- 高质量：Gulp的每个插件只完成一个功能，这也是Unix的设计原则之一，各个功能通过流进行整合并完成复杂的任务。例如：Grunt的imagemin插件不仅压缩图片，同时还包括缓存功能。他表示，在Gulp中，缓存是另一个插件，可以被别的插件使用，这样就促进了插件的可重用性。目前官方列出的有673个插件。

Gulp示例

```
var gulp = require('gulp');
var jshint = require('gulp-jshint');
var concat = require('gulp-concat');
var rename = require('gulp-rename');
var uglify = require('gulp-uglify');

// Lint JS
gulp.task('lint', function() {
return gulp.src('src/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'));
});

// Concat & Minify JS
gulp.task('minify', function(){
return gulp.src('src/*.js')
    .pipe(concat('all.js'))
    .pipe(gulp.dest('dist'))
    .pipe(rename('all.min.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist'));
});

// Watch Our Files
gulp.task('watch', function() {
gulp.watch('src/*.js', ['lint', 'minify']);
});

// Default
gulp.task('default', ['lint', 'minify', 'watch']);
```

参考资料

- gulp
- gulp fiction: 可视化配置gulp工作流程。
- 前端工程的构建工具对比 Gulp vs Grunt
- gulp-cheatsheet: A cheatsheet for gulp.js

gulp插件选择

使用的插件

无依赖的

- del: Delete files/folders using globs.
- opn: A better node-open. Opens stuff like websites, files, executables. Cross-platform.
- psi: PageSpeed Insights for Node.
- wiredep: Wire Bower dependencies to your source code.
- browser-sync: Live CSS Reload & Browser Syncing.
- require-dir: Helper to require() directories.
- run-sequence: Run a series of dependent gulp tasks in order.

gulp辅助

- gulp: The streaming build system.
- gulp-if: Conditionally run a task.
- gulp-cache: A cache proxy task for Gulp.
- gulp-changed: Only pass through changed files.
- gulp-filter: Filter files in a vinyl stream.
- gulp-replace: A string replace plugin for gulp.
- gulp-flatten: remove or replace relative path for files.
- gulp-useref: Parse build blocks in HTML files to replace references to non-optimized scripts or stylesheets.
- gulp-load-plugins: Automatically load any gulp plugins in your package.json.

gulp plugins for css

- gulp-autoprefixer: gulp-autoprefixer.
- gulp-uncss: Remove unused CSS selectors.
- gulp-csso: Minify CSS with CSSO.
- gulp-less: Less for Gulp.

gulp plugins for scripts

- gulp-jshint: JSHint plugin for gulp.
- jshint-stylish: Stylish reporter for JSHint.

gulp plugins for compress

- gulp-concat: Concatenates files.
- gulp-imagemin: Minify PNG, JPEG, GIF and SVG images.
- gulp-minify-html: Minify html with minimize.
- gulp-minify-css: Minify css with clean-css.
- gulp-uglify: Minify files with UglifyJS.
- gulp-size: Display the size of your project.

测试相关

- karma 自动测试插件

- karma-Chrome-launcher: 调用Chrome进行测试

待选

- gulp-livereload
- gulp-autowatch

generator-gulp-webapp

```
"devDependencies": {
    "apache-server-configs": "^2.7.1",
    "connect": "^3.0.1",
    "connect-livereload": "^0.4.0",
    "del": "^0.1.0",
    "gulp": "^3.6.0",
    "gulp-autoprefixer": "^0.0.7",
    "gulp-cache": "^0.2.2",
    "gulp-csso": "^0.2.6",
    "gulp-filter": "^0.5.0",
    "gulp-flatten": "^0.0.2",
    "gulp-if": "^1.2.1",
    "gulp-imagemin": "^0.6.0",
    "gulp-jshint": "^1.5.3",
    "gulp-livereload": "^2.0.0",
    "gulp-load-plugins": "^0.5.0",<% if (includeSass) { if (includeBootstrap) { %>
    "gulp-replace": "^0.3.0",<% } %>
    "gulp-ruby-sass": "^0.5.0",
    "gulp-plumber": "^0.6.3",<% } %>
    "gulp-size": "^0.4.0",
    "gulp-uglify": "^0.3.0",
    "gulp-useref": "^0.6.0",
    "jshint-stylish": "^0.2.0",<% if (includeBootstrap && includeSass) { %>
    "lazypipe": "^0.2.1",<% } %>
    "main-bower-files": "^1.0.1",
    "opn": "^0.1.1",
    "serve-index": "^1.1.4",
    "serve-static": "^1.4.0",
    "wiredep": "^1.4.3"
  }
```

web-starter-kit-master

```
"devDependencies": {
    "apache-server-configs": "^2.7.1",
    "browser-sync": "^1.3.0",
    "del": "^0.1.2",
    "gulp": "^3.8.5",
    "gulp-autoprefixer": "^0.0.8",
    "gulp-cache": "^0.2.2",
    "gulp-changed": "^1.0.0",
    "gulp-csso": "^0.2.9",
    "gulp-flatten": "^0.0.2",
    "gulp-if": "^1.2.1",
    "gulp-imagemin": "^1.0.0",
    "gulp-jshint": "^1.6.3",
    "gulp-load-plugins": "^0.5.3",
    "gulp-minify-html": "^0.1.4",
    "gulp-replace": "^0.4.0",
    "gulp-ruby-sass": "^0.7.1",
    "gulp-size": "^1.0.0",
    "gulp-uglify": "^0.3.1",
    "gulp-uncss": "^0.4.5",
    "gulp-useref": "^0.6.0",
    "jshint-stylish": "^0.4.0",
    "opn": "^1.0.0",
    "psi": "^0.1.2",
```

```
    "require-dir": "^0.1.0",
    "run-sequence": "^0.3.6"
  }
```

yeoman参考案例

generator-generator是用来生成模板的模板。在Yeoman Discover

Yeoman官方

- Yeoman on Github
- generator-webapp
- generator-gulp-webapp

非官方的

- web-starter-kit

SAP

什么是SAP

“A single-page application (SPA), is a web application or web site that fits on a single web page with the goal of providing a more fluid user experience akin to a desktop application.”

参考资料

- 构建单页Web应用
- 移动Web单页应用开发实践——页面结构化

单页面SEO解决方案

单页应用实际是把视图（View）渲染从Server交给浏览器，Server只提供JSON格式数据，视图和内容都是通过本地JavaScript来组织和渲染。而搜索搜索引擎抓取的内容，需要有完整的HTML和内容，单页应用架构的站点，并不能很好的支持搜索。

路由与状态的管理

传统的页面型产品是不存在这个问题的，因为它就是以页面为单位的，也有的时候，服务端路由处理了这一切。但是在单页应用中，这成为了问题，因为我们只有一个页面，界面上的各种功能区块是动态生成的。所以我们要通过对路由的管理，来实现这样的功能。

`#` 号在浏览器的URL中是一个锚点，在当前页改变#号的参数，页面会跳转到锚点所在的位置，通过JavaScript我们可以获取到 `#` 号后的参数, 改变 `#` 号后的参数，页面并不会重载，于是大多数的单页架构网站，都在URL中采用 `#` 号来作为当前视图的URL地址。

prerender.io

万金油的 prerender.io.也就是根据用户请求的UA类型,普通用户请求，按照正常流程走.而蜘蛛爬虫之类的用户请求,采用由prerender执行页面js后生成的静态页面发送给爬虫.从而达到通用的单页面seo.不关乎于框架.(注意游戏直播的鼻祖 twitch也是采用prerender.io的)

参考资料

- 单页应用SEO浅谈

开发无框架单页面应用

参考资料

- 开发无框架单页面应用 — 老码农的祖传秘方
- 项目示例
- 别再用JavaScript框架了
- Single page apps in depth

可伸缩的同构Javascript代码

先花点时间想想你是有多么频繁地听到“Model-View-Controller”（MVC）这词儿，但你真正明白它的意义吗？在较高层次上而言，它是指在一个基于图像系统（非光栅化图像，比如游戏）以展示为主的应用中对功能的关注点分离（separation of concerns）。进一步看，它就是一堆表示不同事物的专有名词。过去，许多开发者社区都创造了各自的MVC解决方案，它们都能很好地应对流行的案例，并且在一步一步地发展。最好的例子就是Ruby和Python社区以及它们基于MVC架构的Rails与Django框架。

MVC模式已经被其它语言所接受，比如Java，Ruby和Python。但是对于Node.js而言还不够好，其中的一个原因就是：Javascript现在是一个同构的语言了。同构的意义就在于任何一段代码（当然有些特殊代码例外）都能同时跑在客户端与服务器端。从表面上讲，这个看似无害的特性带来了一系列当前的MVC模式无法解决的挑战。在这篇文章中我们会探寻目前存在一些的模式，看看它们都是怎样实现的，同时关注不同的语言及环境。另外也谈谈它们为什么对于真正同构的Javascript而言还不够好。在最后，我们会了解一种全新的模式：Resource-View-Presenter。

题要

设计模式在应用开发中至关重要。它们概述、封装了应用程序及其环境中值得关注的地方。在浏览器与服务器之间这些关注点差异很大：

- 视图是短暂的（如在服务器上）还是长期存在的（如在浏览器上）？
- 视图是否能跨案例或场景复用？
- 视图是否该被应用特定的标签标记？
- 一堆堆的业务逻辑应该放哪里？（在Model中还是在Controller中？）
- 应用的状态应该如何持久化和访问？

参考资料

- 可伸缩的同构Javascript代码

PJAX

参考资料

- HTML5 History API + Ajax (Pjax) 实现友好的局部刷新
- pjax 是如何工作的？
- ajax与HTML5 history pushState/replaceState实例

前后端分离

参考资料

- 淘宝前后端分离实践

一个简单粗暴的前后端分离方案

参考资料

- 一个简单粗暴的前后端分离方案

User Interface

UI(User Interface), 用户界面。针对界面美观，操作逻辑以及人机交互的设计。

关于Retina

Device Pixels(设备像素)

一个设备像素（或者称为物理像素）是显示器上最小的物理显示单元。在操作系统的调度下，每一个设备像素都有自己的颜色值和亮度值。

PPI(pixels per inch)

屏幕密度指的是单位面积里物理像素的数量，通常以PPI(pixels per inch)为单位。苹果公司为它的双倍屏幕密度的显示器（double-density displays）创造了一个新词“Retina”，声称在正常的观看距离下，人眼无法在Retina显示器上分辨出单独的像素。

与设备无关的像素（DIPs,device-independent pixels）

CSS pixel是浏览器使用的抽象单位，用来精确的、统一的绘制网页内容。通常，CSS pixels被称为与设备无关的像素（DIPs,device-independent pixels）。在标准密度显示器（standard-density displays）上，1 CSS pixel对应一个物理像素。

物理像素与DIPs的比例

物理像素与CSS pixel 的比率可以通过媒体查询的device-pixel-ratio来检测（device-pixel-ratio兼容性）。也可以通过javascript的window.devicePixelRatio来获取该比率。

Bitmap Pixels（位图像素）

一个位图像素是栅格图像（也就是位图，png、jpg、gif等等）最小的数据单元。每一个位图像素都包含着该如何显示自己的信息，例如显示位置、颜色值等。一些图片格式还包含额外的数据，例如透明度。

除了自身的分辨率外，图片在网页上还有一个抽象的尺寸，通过CSS pixels来定义。浏览器在渲染的过程中，会根据图片的CSS高度和宽度来压缩或是拉伸图片。

当一个位图以原尺寸展示在标准密度显示器上时，一位图像素对应一个物理像素，就是无失真显示。而在Retina显示器上，为了保证同样的物理尺寸，需要用四倍的像素来展示，但由于单个位图像素已经无法再进一步分割，只能就近取色，导致图片变虚。

解决方案

HTML

img标签创建的时候，写上一倍大小的宽高。

CSS

通过background属性进行控制:

```
.image {
```

```
  background-image: url(example@2x.png);
  background-size: 200px 300px;
  /* 或者是用 background-size: contain; */
  height: 300px;
  width: 200px;
}
```

JavaScript

像素密度可以通过javascript的window.devicePixelRatio来查询(注意：不是所有浏览器都支持devicePixelRatio)。一旦检测到高密度显示器，你就可以用高质量图片替换普通图片：

```
$(document).ready(function(){
  if (window.devicePixelRatio > 1) {
    var lowresImages = $('img');
    images.each(function(i) {
      var lowres = $(this).attr('src');
      var highres = lowres.replace(".", "@2x.");
      $(this).attr('src', highres);
    });
  }
});
```

SVG(Scalable Vector Graphics)

由于位图本身固有的性质，不可能无限制的缩放。而恰恰这是矢量图的优势所在。

Icon Fonts

Twitter的 bootstrap使Icon Fonts 更加的流行，该技术是通过@font-face引入基于icon的字体来代替位图icon，这样icon就不再受分辨率影响。用纯色icon代替字母的web Fonts，可以用CSS来调整样式，就像网页里其它文本一样。

参考资料

- 【译】走向Retina Web

响应式图片

参考资料

- 实战响应式图片

响应式字体

px, em ,rem这是web开发中常用的几种字体单位。

关于字号单位

对于webapp上文字用什么单位的问题，一直以来都是让我们csser头疼的问题，公说公有理，婆说婆有理。有人说px好，有人说em自适应，有的说百分比牛逼，rem文字出来就跟风说目前最好的就是rem单位。不管是什么说，我们还是要实地捣腾一下。

px

px是指相对于自身的字体大小的单位。

px像素单位是针对电脑屏幕来说的一个单位，对于桌面上来说，衡量屏幕尺寸的就是分辨率了，1920*1280的分辨率屏幕，横向就是1920像素，纵向1280个像素点（除高清屏幕外），那我们设置一个字体样式 font-size:12px 计算得出来的应该是相对于电脑屏幕分辨率的12个单位长度，所以有时候我们会陷入一个误区：px像素单位是一个绝对长度单位，但是其实它也是一个相对单位长度，它相对它的显示设备分辨率。

em

em（font size of the element）是指相对于父元素的字体大小的单位。

1em默认为16px。

rem

rem（font size of the root element）是指相对于根元素的字体大小的单位。简单的说它就是一个相对单位。

1rem默认为10px。

参考资料

- web app变革之rem
- 关于webapp中的文字单位的一些捣腾
- 7个你可能不认识的CSS单位

中文字体

我们在日常需求中，经常会碰到视觉设计师对某个中文字体效果非常坚持的情况，因为页面是否高大上，字体选择是很重要的一个因素，选择合适的字体可以让页面更优雅。

对于特殊字体，目前使用最多的情况还是转换为图片显示。

使用图片

优点：还原度高

缺点：

- 制作与维护成本很高。切图繁琐、高清屏适配繁琐、合并雪碧图更繁琐，后期修改更加繁琐
- 用户体验差。导致网页不支持选中、复制、搜索、翻译、矢量缩放，也会影响视障用户使用读屏器操作网页
- 带来更多带宽消耗。导出的图片体积随着文本面积增加，且字形无法重复利用，这消耗着大量的服务器资源

WebFont

WebFont技术提供了在网页使用特殊字体的可能，从而避免用图片的方法。它的实现方法是通过CSS的@font-face引入字体。很多互联网公司已经率先采用了这种方法，比如Apple官网就是采用了自己的字体。Google也推出了免费的WebFont云托管服务，在国外网站自定义字体得到很好的应用。

中文webFont的一个选择是Adobe与Google所领导开发的开源字体——思源字体。

中文WebFont的困境

- 中文字体体积大
- 浏览器支持,不同浏览器支持的字体格式不同,主要格式有四种: ttf, woff, eot, svg

参考资料

- 设计师的春天：中文WebFont解决方案Font-Spider(字蛛)

移动端字体

在移动设备上要面临与生俱来的挑战：空间有限，环境光通常比较微弱。下面列出一些技巧：

留足空间

与普遍观点恰好相反，字体并非屏幕上弯弯曲曲的线条排列；它主要在于周围和相互间的空间。 字母本身对字体的影响，与构成它的空间相比，要小得多。

要理解这一点，了解字体从何而来很有帮助：字母o（还有b、c、p等等）中间的圆孔被称作"凹槽"。在最原始的印刷机上，铅字由金属雕刻而成，这些凹槽来自雕刻成型、排列在盘中的金属活字。第一个字体设计师所处理的模具，实际上并不能用于印刷。字母本身对字体的影响，与构成它的空间相比，要小得多。 谈到层次时，我们通常指的是h1到p，有时候还会到h6。但另外还有一种层次在影响着行或段落的视觉流，这是特殊的层次：字母间距小于字间距，字间距小于行间距，以此类推。 要在移动端创造最佳易读性，尤其要注意这些特殊层次，这些格式塔式的词语、行、段落的文字组合，在自然光环境下同样至关重要。

行宽

行宽是一行文字的长度。或者确切的说，是一行文字的理想长度，因为很难让每一行都精确吻合。

众所周知，舒适阅读的理想行宽是65个字符左右。行宽产生的物理长度，取决于字体的设计、字间距（见下文）和你使用的具体文字。本文开篇的65个字符（译者注：此处请参见英文原文），用PT Serif字体是26.875em宽，用Open Sans是28.4375em宽，用Ubuntu字体是27.3125em宽。如果再加入斜体、大小写和一大堆其他字体细节，还会有更大的差异。 在桌面端浏览器中，65个字符很难触及边缘，但在移动设备上，65个字符（如果至少大到看得清）会超出浏览器的边界。所以，在移动设备上，你必须得缩减行宽。

移动端并没有普遍认可的行宽标准。不过传统上，报纸或杂志上每一个窄列都会趋向于39个字符。鉴于这个理想行宽已经经历了数个世纪的考验，它在移动端字体上也运转良好。

宽松行距、紧凑行距

行距是行之间的空间，行距太紧凑，会让视线难以从行尾扫视到下一行首。行距太宽松，字间距会开始形成队列，产生了我们通常意义上的河流，阻断了行的视觉流。

行距的标准通常是1.4em，但以我的经验，这对于屏幕来说太紧凑了：在屏幕上表现良好的字体都有一个关键特征——大的凹槽，大凹槽需要更大一些的行距来保持空间层次。

反过来，更短的行宽需要更小的行距。所以你可能需要将桌面端的行距设得宽松点，同时记得将移动端的设置得紧凑些。

找到最佳状态

所有字体至少都有一种最佳状态，在屏幕上展现最佳的尺寸，还有在浏览器中最能保持字形的抗锯齿选项。

最佳状态下，多数笔画通常都能排列在像素网格中——像素字体，如果你还记得的话，那些字体仅仅在字号调整到最佳状态下才有效。

将字体设为最佳状态能形成更强烈的对比。为移动端设计时，对比尤其重要，因为户外的强光可能分散注意。

你会发现，微调行距会使每行脱离完美像素匹配。我觉得，在移动设备屏幕上，对比的重要性胜过行距。所以如果你不得不

在行距上妥协，来保持每行契合像素网格，那就这么做吧。

通常设计师通过基线网格来排列文字。但在移动设备上，我们需要使用x高度来代替（x高度顾名思义，就是小写字母x的高度）。从易读性研究中，我们知道大脑识别的是文字顶部，而不是底部。所以要成就更加平顺的视觉流，我们要确保字符顶部最契合像素网格。

不要忽视起伏边

起伏边是一段文字的边缘。你读的多数内容是居左对齐的（至少对于拉丁语系而言），导致右边沿参差不齐。

当视线从行尾跳至下一行首时，大脑最好要能判断出下一次跳跃的角度和距离。把每次跳跃都想象成跑过跳板，如果间距保持一致，就会快很多。因此，文字左侧边缘应该是平的，每行从同一个地方开始（对于从右至左的语言，恰好相反）。

因此你绝不应该将两三行以上的文字居中对齐。

通常文字会设置成两端对齐，这意味着每行文字所占空间相等，所以两侧都不会有起伏边。我怀疑两端对齐的流行和响应式设计有关，它教设计师们以块状形态思考。两端对齐的文字产生的留白不统一。最糟的情况会导致一行中只有几个字，相当不协调。更窄的行宽会加重两端对齐的问题，所以两端对齐的文字在移动端是难以阅读的。

减少反差

增强文字与背景对比的同时，我们也要减少不同层次文字间的反差。

在移动端，实际可见的文字更少，所以反差被放大了。

其原因是我们的大脑基于环境来判断重要性。在桌面端，标题可能是正文字号的两倍甚至三倍，因为屏幕上有更多文字，所以这是有效的。在移动端，实际可见的文字更少，所以反差被放大了。

多数设计师使用斐波那契数列式的字号组合。在移动端，应该缩小比率来减少字号间的反差。比如，如果你使用黄金比例1.618与字号相乘。在移动端，应该用更小的比例**1.382**来替代。

按比例调整字间距

为移动端调整字号时，我们要意识到字间距发生了必要的变化。

（先说一句，不应该调整固有字距。固有字距是两个字母相互组合时的距离，使它们的间距与其他字母间距在视觉上统一。创作字体时，就纳入了固有字距的考量，这个过程可能要花上数月。如果你选用了一款专业的字体，它的固有字距就是合适的，如果你觉得不对，请换一个字体。）

字间距并不是固有字距。字间距是字体中应用在所有字符上的间距。通常你也不应该调整字间距。

大字号是个例外，拿标题和小号文字（比如脚注）举例。大号文字需要减少字间距，小号文字需要增加字间距。前者是考虑到分组，后者则是为了增强对比。如果你在调整标题，或是用了通常字间距紧密的艺术字体，缩小时可能就需要把字间距放开一点。

总结

字体是一门工艺，设计师终其一生都在精心打磨。的确如此，因为每个文字、每种字体和每项技术都带来了新的挑战。没有一成不变的普适规律。 假如你追求易读性，要牢记三条原则：行内的视觉流要平顺，空间层级要清晰，要有足够的对比。这尤其适用于移动端页面。 没有不可撼动的规则，全凭你双眼决断。不过本文的指南可以作为理想的出发点，让你在移动设备上优美地排列文字。

参考资料

- 移动端字体7准则

响应式设计原则

参考资料

- 响应式布局的三大要点
- 两大设计模式 塑造高可读性的网页布局

中文排版的规范

参考资料

- W3C: 中文排版规范

User Experience

用户体验，User experience.

参考资料

- User experience

页面滚动条出现时不跳动

当前web届，绝大多数的页面间布局都是水平居中布局，主体定个宽度, 然后水平居中。然而，这种布局有一个存在一个影响用户体验的隐患。应该都知道，现代浏览器滚动条默认是overflow:auto类型的，也就是如果尺寸不足一屏，没有滚动条；超出，出现滚动条。

结果就会造成出现滚动条的时候，页面会向左侧跳动一下，这个体验上是会有一个微小的影响的。

当前的解决方案

高度不确定的

例如，新浪微博，使用`overflow-y: scroll;`。

缺点:在页面高度较小的时候，依然会保留一个丑陋的灰色的滚动栏，这其实又回到了IE当道的旧社会时代。现代浏览器做的那些默认视觉优化岂不是白费了，想想就好痛心。

高度确定的

使用CSS把页面尺寸布局骨架搭好，再在里面吐数据。于是，要么没有滚动条，要么滚动条直接出现。不会出现跳动。

缺点：只适合一些特殊的定制性很强的页面。你说像知乎这样子，高度随内容而定的页面，显然就无法驾驭.

更好的解决方案

使用CSS3计算calc和vw单位巧妙实现滚动条出现页面不跳动。代码如下：

```
.wrap-outer {
    margin-left: calc(100vw - 100%);
}

/* or */

.wrap-outer {
    padding-left: calc(100vw - 100%);
}
```

说明:

- .wrap-outer指的是居中定宽主体的父级，如果没有，创建一个（使用主体也是可以实现类似效果，不过本着宽度分离原则，不推荐）；
- calc是CSS3中的计算，IE10+浏览器支持，IE9浏览器基本支持(不能用在background-position上)；
- 100vw相对于浏览器的window.innerWidth，是浏览器的内部宽度，注意，滚动条宽度也计算在内！而100%是可用宽度，是不含滚动条的宽度。
- calc(100vw - 100%)就是浏览器滚动条的宽度大小（如果有，如果没有滚动条则是0）！左右都有一个滚动条宽度（或都是0）被占用，主体内容就可以永远居中浏览器啦，从而没有任何跳动！

兼容性:IE9+以及其他现代浏览器。

缺点:窄屏幕宽度下的处理。浏览器宽度比较小的时候，左侧留的白明显与右边多，说不定会显得有点傻。那么就在响应式的时候进行如上处理。

参考资料

- 小tip:纯CSS让overflow:auto页面滚动条出现时不跳动
- fix-jumping-scrollbar

使用渐进式 JPEG 来提升用户体验

今天才认识到原来JPEG文件有两种保存方式，分别是Baseline JPEG（标准型）和Progressive JPEG（渐进式）。两种格式有相同尺寸以及图像数据，扩展名也是相同的，唯一的区别是二者显示的方式不同。

Baseline JPEG

这种类型的JPEG文件存储方式是按从上到下的扫描方式，把每一行顺序的保存在JPEG文件中。打开这个文件显示它的内容时，数据将按照存储时的顺序从上到下一行一行的被显示出来，直到所有的数据都被读完，就完成了整张图片的显示。如果文件较大或者网络下载速度较慢，那么就会看到图片被一行行加载的效果，这种格式的JPEG没有什么优点，因此，一般都推荐使用Progressive JPEG。

Progressive JPEG

和Baseline一遍扫描不同，Progressive JPEG文件包含多次扫描，这些扫描顺寻的存储在JPEG文件中。打开文件过程中，会先显示整个图片的模糊轮廓，随着扫描次数的增加，图片变得越来越清晰。这种格式的主要优点是在网络较慢的情况下，可以看到图片的轮廓知道正在加载的图片大概是什么。在一些网站打开较大图片时，你就会注意到这种技术。

渐进式图片带来的好处是可以让用户在没有下载完图片就可以看到最终图像的大致轮廓，一定程度上可以提升用户体验。（瀑布留的网站建议还是使用标准型的）

另外渐进式的图片的大小并不会和基本的图片大小相差很多，有时候可能会比基本图片更小。渐进式的图片的缺点就是吃用户的CPU和内存，不过对于现在的电脑来说这点图片的计算并不算什么。

如何保存Progressive JPEG图片

PhotoShop

在photoshop中有“存储为web所用格式”，打开后选择“连续”就是渐进式JPEG。

参考资料

- 使用渐进式 JPEG 来提升用户体验

动画相关

介绍一些动画相关的内容，包括实例，原理与实现。

跟动画有关的数学和物理公式

角度与弧度互转

```
radians=degrees*Math.PI/180
degrees=radians*180/Math.PI
```

朝鼠标指针（或任意一点）旋转

```
dx=mouse.x-object.x;
dy=mouse.y-boject.y;
object.rotation=Math.atan2(dy,dx)*180/Math.PI;
```

创建波

```
(function(){
  window.requestAnimationFrame(drawFrame,canvas);
  value=center+Math.sin(angle)+range;
  angle+=speed;
}());
```

创建圆形

```
(function(){
  window.requestAnimationFrame(drawFrame,canvas);
  xposition=centerX + Math.cos(angle) * radius;
  yposition=center + Math.sin(angle) * radius;
  angle += speed;
}());
```

创建椭圆

```
(function(){
  window.requestAnimationFrame(drawFrame,canvas);
  xposition=centerX + Math.cos(angle) * radiusX;
  yposition=center + Math.sin(angle) * radiusY;
  angle += speed;
}());
```

获得两点间的距离

```
dx = x2 - x1;
dy = y2 - y1;
dist = Math.sqrt(dx * dx + dy * dy);
```

绘制一条穿越某个点的曲线

```
x1 = xt * 2 - (x0 + x2) / 2;
y1 = yt * 2 - (y0 + y2) / 2;
context.moveTo(x0, y0);
context.quadraticCurveTo(x1, y1, x2, y2);
```

将角速度分解为x、y轴上的速度向量

```
vx = speed * Math.cos(angle);
vy = speed * Math.sin(angle);
```

将角加速度（作用于物体上的力）分解为x、y轴上的加速度

```
ax = force * Math.cos(angle);
ay = force * Math.sin(agnle);
```

将加速度加入速度向量

```
vx += ax;
vy += ay;
```

将速度向量加入位置坐标

```
object.x += vx;
object.y += vy;
```

移除越界物体

```
if(object.x - object.width /2 > right ||
   object.x + object.width /2 < left ||
   object.y - object.height /2 > bottom ||
   object.y + object.height /2 < top){

}
```

重置越界物体

```
if(object.x - object.width /2 > right ||
   object.x + object.width /2 < left ||
   object.y - object.height /2 > bottom ||
   object.y + object.height /2 < top){

}
```

屏幕环绕越界物体

```
if(object.x - object.width /2 > right){
  object.x = left - object.width / 2;
}else if(object.x + object.width /2 < left){
  object.x = right + object.width /2;
}
if(object.y - object.height / 2 > bottom){
  object.y = top -  object.height / 2;
}else if(object.y + object.height / 2 < top){
  object.y = bottom + object.height /2;
}
```

应用摩擦力（正确方法）

```
speed = Math.sqrt(vx * vx + vy * vy);
angle = Math.atan2(vy,vx);
if(speed > friction){
  speed -= friction;
}else{
  speed = 0;
}
vx = Math.cos(angle) * speed;
vy = Math.sin(angle) * speed;
```

应用摩擦力（简便方法）

```
vx *= friction;
vy *= friction;
```

简单缓动

```
object.x += (targetX - object.x) * easing;
object.y += (targetY - object.y) * easing;
```

简单弹动

```
vx += (targetX - object.x) * spring;
vy += (targetY - object.y) * spring;
object.x += (vx *= friction);
object.y += (vy *= friction);
```

有偏移量的弹动

```
var dx = object.x - fixedX,
    dy = object.y - fixedY,
    angle = Math.atan2(dy,dx),
    targetX = fixedX + Math.cos(angle) * springLength,
```

```
    targetY = fixedX + Math.sin(angle) * springLength;
```

基于距离的碰撞检测

```
var dx = objectB.x - objectA.x,
    dy = objectB.y - objectB.y,
    dist = Math.sqrt(dx * dx + dy * dy);

if(dist < objectA.radius + objectB.radius){

}
```

多物理碰撞检测

```
objects.forEach(function(objectA, i){
  for(var j = i + 1; j < objects.length; j++){
    var objectB = objects[j];
    //执行碰撞检测，在objectA和objectB之间。
  }
});
```

坐标旋转

```
x1 = x * Math.cos(rotation) - y * Math.sin(rotation);
y1 = y * Math.cos(rotation) + x * Math.sin(rotation);
```

反向坐标旋转

```
x1 = x * Math.cos(rotation) + y * Math.sin(rotation);
y1 = y * Math.cos(rotation) - x * Math.sin(rotation);
```

动量守恒

```
var vxTotal = vx0 -vx1;
vx0 = ((ball0.mass -ball1.mass) * vx0 + 2 * ball1.mass * vx1) / (ball0.mass + ball1.mass);
vx1 = vxTotal + vx0;
```

万有引力

```
function gravitate(partA, partB){
  var dx = partB.x - partA.x;
      dy = partB.y - partA.y;
      distSQ =  dx * dx + dy * dy;
      dist = Math.sqrt(distSQ);
      force = partA.mass * partB.mass / distSQ;
      ax = force * dx /dist;
      ay = force * dy / dist;
```

```
    partA.vx += ax / partA.mass;
    partA.vy += ax / partA.mass;
    partB.vx -= ax / partB.mass;
    partB.vy -= ax / partB.mass;
  }
```

余弦定理

```
var A = Math.acos((b * b + c * c - a * a) / (2 * b * c));
var B = Math.acos((a * a + c * c - b * b) / (2 * a * c));
var C = Math.acos((a * a + b * b - c * c) / (2 * a * b));
```

基本透视图

```
scale = fl / (fl + zpos);
object.scaleX = object.scaleY = scale;
object.alpha = scale;
object.x = vanishingPointX + xpos * scale;
object.y = vanishingPointY + ypos * scale;
```

Z排序

```
function zSort(a, b){
  return (b.zpos - a.pos);
}
objects.sort(zsort);
```

坐标旋转

```
x1 = xpos * cos(angleZ) - ypos * sin(angleZ);
y1 = ypos * cos(angleZ) - xpos * sin(angleZ);

x1 = xpos * cos(angleY) - zpos * sin(angleY);
z1 = zpos * cos(angleY) + xpos * sin(angleY);

y1 = ypos * cos(angleX) - zpos * sin(angleX);
z1 = zpos * cos(angleX) + ypos * sin(angleX);
```

三维距离

```
dist = Math.sqrt(dx * dx + dy * dy + dz * dz);
```

参考资料

- A Quick Look Into The Math Of Animations With JavaScript

缓动的原理与实现

动画就是以一定的频率去改变元素的属性，使之运动起来，最普通的动画就是匀速的动画，每次增加固定的值。缓动就是用来修改每次增加的值，让其按照不规律的方式增加，实现动画的变化。

程序实现缓动

没有加速度的线性运动

数学公式为：`f(x)=x`，代码如下：

```
AnimationTimer.makeLinear = function () {
   return function (percentComplete) {
      return percentComplete;
   };
};
```

逐渐加速的缓入运动

数学公式为：`f(x)=x^2`，代码如下：

```
AnimationTimer.makeEaseIn = function (strength) {
   return function (percentComplete) {
      return Math.pow(percentComplete, strength*2);
   };
};
```

逐渐减速的缓出运动

数学公式为：`f(x)=1-(1-x)^2`，代码如下：

```
AnimationTimer.makeEaseOut = function (strength) {
   return function (percentComplete) {
      return 1 - Math.pow(1 - percentComplete, strength*2);
   };
};
```

缓入缓出运动

数学公式为：`f(x)=x-sin(x*2π)/(2π)`，代码如下：

```
AnimationTimer.makeEaseInOut = function () {
   return function (percentComplete) {
      return percentComplete - Math.sin(percentComplete*2*Math.PI) / (2*Math.PI);
   };
};
```

弹簧运动

数学公式为：`f(x)=(1-cos(x*Npasses * π) * (1-π))+x`，Npassed表示运动物体穿越中轴的次数。 代码如下：

```
AnimationTimer.makeElastic = function (passes) {
   passes = passes || 3;
   return function (percentComplete) {
       return ((1-Math.cos(percentComplete * Math.PI * passes)) *
               (1 - percentComplete)) + percentComplete;
   };
};

## 弹跳运动
Nbounces表示运动物体被弹起的总次数，弹起的次数为偶数的时候，数学公式为：
```

f(x)=(1=cos(x *Nbounces* π) * (1-π))+x

```
弹起的次数为奇数的时候，数学公式为：
```

f(x)=2-(((1-cos(x π Nbounces)) * (1-x)+x)

```
代码如下：
```

AnimationTimer.makeBounce = function (bounces) { var fn = AnimationTimer.makeElastic(bounces); return function (percentComplete) { percentComplete = fn(percentComplete); return percentComplete <= 1 ? percentComplete : 2-percentComplete; }; }; ```

参考资料

- jQuery Easing Plugin

常用动画与缓动

单页面

单页面不同锚点之间的跳转，常用的缓动效果是，慢->快->慢。对应的缓动是: `easeInOutQuart` 。

hover效果

mouseenter的时候立即显示, mouseleave的时候，做一个延时。体验上会更好。

参考资料

- 视“差”滚动浅析
- 动效设计的物理法则

SVG动画

制作的过程，一般都是先用绘图工具绘制出来然后再加的动画。用ai或其他矢量工具画，导出svg。

库文件

- vivus: 可控制svg绘制动画。
- Bonsai
- Velocity.js:Velocity is an animation engine with the same API as jQuery's $.animate(). It works with and without jQuery.
- Raphaël: Raphaël is a small JavaScript library that should simplify your work with vector graphics on the web.
- SnapSVG: Snap was written entirely from scratch by the author of Raphaël (Dmitry Baranovskiy), and is designed specifically for modern browsers (IE9 and up, Safari, Chrome, Firefox, and Opera).
- Walkway: Walkway 支持3种方式, path, line 和 用polyline来画的svg线。它提供了一个很好的例子，绘制了一个PlayStation 的集合动画。
- SVG.js
- CHARTIST.JS : SIMPLE RESPONSIVE CHARTS

参考资料

- 如何绘制SVG格式的图标
- svg动画的实现

原理性质

讲述一些底层的东西。

参考资料

- awesome-wpo

单线程的Javascript

浏览器的内核是多线程的，它们在内核制控下相互配合以保持同步，一个浏览器至少实现三个常驻线程：

- javascript引擎线程 javascript引擎是基于事件驱动单线程执行的，JS引擎一直等待着任务队列中任务的到来，然后加以处理，浏览器无论什么时候都只有一个JS线程在运行JS程序。
- GUI渲染线程 GUI渲染线程负责渲染浏览器界面，当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时,该线程就会执行。但需要注意GUI渲染线程与JS引擎是互斥的，当JS引擎执行时GUI线程会被挂起，GUI更新会被保存在一个队列中等到JS引擎空闲时立即被执行。
- 浏览器事件触发线程 事件触发线程，当一个事件被触发时该线程会把事件添加到待处理队列的队尾，等待JS引擎的处理。这些事件可来自JavaScript引擎当前执行的代码块如setTimeOut、也可来自浏览器内核的其他线程如鼠标点击、AJAX异步请求等，但由于JS的单线程关系所有这些事件都得排队等待JS引擎处理。（当线程中没有执行任何同步代码的前提下才会执行异步代码）

单线程的证明

下面的代码，当while执行时候，setTimeout永远不会执行。

```
var isEnd = true;
     window.setTimeout(function () {
         isEnd = false;//1s后，改变isEnd的值
     }, 1000);
     //这个while永远的占用了js线程，所以setTimeout里面的函数永远不会执行
     while (isEnd);
     //alert也永远不会弹出
     alert('end');
```

参考资料

- JavaScript的计时器的工作原理

v8引擎

JavaScript在V8引擎中是如何工作的？

组成部分

- 一个基本的编译器（basecompiler），在你的代码运行之前，它会分析你的JavaScript代码并且生成本地的机器码，而不是通过字节码的方式来运行，也不是简单地解释它。这种机器码起初是没有被高度优化的。
- V8通过对象模型（objectmodel）来表达你的对象。对象是在JavaScript中是以关联数组的方式呈现的，但是在V8引擎中，它们是通过隐藏类（hiddenclasses）的方式来表示的。这是一种可以优化查找的内部类型机制（internaltypesystem）。
- 一个运行期剖析器（runtimeprofiler），它会监视正在运行的系统，并且标识出“热点”函数（“hot”function），也就是那些最后会花费大量运行时间的代码。
- 一个优化编译器（optimizingcompiler），重新编译并优化运行期剖析器所标识“热点”代码，然后执行优化，例如，把代码进行内联化（inlining）（也就是在函数被调用的地方用函数主体去取代）。
- V8引擎支持逆优化（deoptimization），意味着如果优化编译器发现在某些假定的情况下，把一些已经优化的代码进行了过度的优化，它就会把它们从生成的代码中抽离出来。
- V8拥有垃圾回收器。理解它是如何运作的和理解如何优化你的JavaScript代码同等重要。

垃圾回收

垃圾回收是一种内存管理机制。垃圾回收器的概念是，它会尝试去重新分配已经不需要的对象所占据的内存空间。在如JavaScript拥有垃圾回收机制的语言中，如果你的程序中仍然存在指向一个对象的引用，那么该对象将不会被回收。

在大多数的情况下，我们没有必要去手动得解除对象的引用（de-referencing）。只要简单地把变量放在它们应该的地方（在理想的情况下，变量应该尽量为局部变量，也就是说，在它们被使用的函数中声明它们，而不是在更外层的作用域），垃圾就能正确地被回收。

参考资料

- 编写快速、高效的JavaScript代码

浏览器渲染

浏览器显示页面的原理

- 获取 HTML 文档及样式表文件
- 解析成对应的树形数据结构
    - DOM tree
    - CSSOM tree
- 计算可见节点形成 render tree
- 计算 DOM 的形状及位置进行布局
- 将每个节点转化为实际像素绘制到视口上（栅格化）

render tree（页面上所显示的最终结果）是由 DOM tree（开发工具中所显示的 HTML 所定义的内容结构）与 CSSOM tree（样式表所定义的规则结构）合并并剔除不可见的节点所形成的，其中不包含如下节点:

- 本身不可见的
    - `<html>`
    - `<head>`
    - `<meta>`
    - `<link>`
    - `<style>`
    - `<script>`
- 设置了 display: none; 样式的

参考资料

- 浏览器的工作原理：新式网络浏览器幕后揭秘
- 开发者需要了解的WebKit
- 理解WebKit和Chromium: HTML解析和DOM
- 前端文摘：深入解析浏览器的幕后工作原理
- 浏览器的渲染原理简介
- 专题：浏览器原理
- 浏览器加载和渲染HTML的顺序以及Gzip的问题
- 从FE的角度上再看输入url后都发生了什么
- 当你在浏览器中输入Google.com并且按下回车之后发生了什么？

JS动画性能

参考资料

- 求索：GSAP的动画快于jQuery吗？为何？

Repaint和Reflow

参考资料

- 翻译：让网络更快一些——最小化浏览器中的回流(reflow)
- 回流与重绘：CSS性能让JavaScript变慢？
- 探讨css中repaint和reflow
- REFLOWS & REPAINTS: CSS PERFORMANCE MAKING YOUR JAVASCRIPT SLOW?
- css Triggers: 查询哪些属性会引起repaint或reflow。
- Google: Make the Web Faster
- 页面重绘和回流以及优化

URL编码与解码

参考资料

- URL编码与解码

性能优化

主要说一说前端的的集成解决方案，目前各自公司都会有自己的一套方式，国外Facebook,国内的baidu做的尤为出色。

编码规范

一方面是代码缩减的问题，另一方面是语法结构的问题。

代码缩减，好像有个文件叫做 `.editorconfig`，能够控制文本的缩减。

代码的语法规范，js提供了一个 `.jshintrc`.

参考资料

- EditConfig
- ESLint
- JSHint

- JavaScript 风格指南/编码规范（Airbnb公司版）
- jQuery编码简洁之道
- CoffeeScript 编码风格指南
- node-style-guide

- bootstrap style
- Standards for developing flexible, durable, and sustainable HTML and CSS
- Front-end Code Standards & Best Practices
- Baidu EFE team specifications
- 支付宝:写样式的更好方式
- frontend-guidelines: github上很受欢迎的一份编码规范, star数超过4000+
- 编写更好的CSS代码

JavaScript代码最佳实践

不要类型转换

JavaScript是动态类型，但如果你想提高速度不要使用该功能。尽量保持变量的类型一致。这也适用于数组，尽管主要是由浏览器都进行了优化,但尽量不要混用不同类型的数组。这就是为何编译成 JavaScript的C/C++代码使用静态类型的原因之一。

字符串与数字类型间相互转换, 一般使用parseInt函数是正确的。

不要重新构造对象

重组对象不便宜， 应该避免它, 不要使用delete运算符。

不要以后再添加属性, 尽量不要在以后再添加属性，最好从一开始就定义对象的架构。这在Firefox中快100%，在Chrome中快89%。

字符串联连

字符串联连是一个非常昂贵的操作，但是应该用什么方法呢？当然不是Array.prototype.join。

+=运算符似乎比+快很多，他们在两种浏览器上比String.prototype.concat和Array.prototype.join都更快。Array.prototype.join是最慢的，符合市场预期。

正确的使用正则表达式

使用RegExp.prototype.exec是没有必要，不是吗？

然而，RegExp.prototype.test和String.prototype.search之间是有性能差异的，让我们来看看哪个方法更快：正则表达式的方法

RegExp.prototype.exec比String.prototype.match快了不少，但他们是不完全一样的东西，它们的区别超出了本文的范围，看这个问答。

RegEx.prototype.test更快，可能是因为它不返回找到匹配的索引。 String.prototype.search应仅用于找到所需的匹配的索引。

然而，你不应该使用正则表达式来查找另一个字符串的位置，你可以使用String.prototype.indexOf方法。

String.prototype.search VS String.prototype.indexOf

另一个有趣的基准是String.prototype.indexOf VS RegExp.prototype.test，我个人预计后者要快，这是在Firefox中发生的事情，但在Chrome中，事实并非如此。 RegExp.prototype.test在Firefox中快32%，而在Chrome中String.prototype.indexOf快33%。在这种情况下，你自己选择喜欢的方式吧。

限制声明/传递变量的范围（作用域）

假如你调用一个函数，浏览器必须做一些所谓的范围查找，它的昂贵程度取决于它要查找多少范围。尽量不要依赖全局/高范围的变量，尽量使局部范围变量，并将它们传递给函数。更少的范围查找，更少的牺牲速度。

这个测试告诉我们，从局部范围内传递和使用变量比从更高的声明范围查找变量快，无论是Chrome和Firefox。

你不需要所有的东西都用**jQuery**

大多数开发者使用jQuery做一些简单的任务，我的意思在一些场合你没有必要使用jQuery，你觉得用$.val()始终是必要的吗？就拿这个例子：

```
$('input').keyup(function() {
    if($(this).val() === 'blah') { ... }
});
```

这是学习如何使用JavaScript修改DOM的最重要原因之一，这样你可以编写更高效的代码。用纯JavaScript100%完成同样的功能100%的速度更快:

```
$('input').keyup(function() {
  if(this.value === 'blah') { ... }
});
```

参考资料

- 编写高质量JavaScript代码的基本要点
- Thinkful: Javascript Best Practices
- 让我们写快速的JavaScript,JS性能优化小窍门
- 编写快速、高效的JavaScript代码
- 编写更加稳定/可读的javascript代码

移动H5前端性能优化指南

移动H5前端性能优化指南 v1.0

PC优化手段在Mobile侧同样适用

在Mobile侧我们提出三秒种渲染完成首屏指标

首屏加载3秒完成或使用Loading

基于联通3G网络平均338KB/s(2.71Mb/s)，所以首屏资源不应超过1014KB

Mobile侧因手机配置原因，除加载外渲染速度也是优化重点



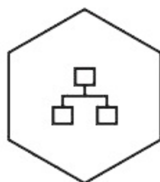
加载优化

1. 合并CSS、JavaScript
2. 合并小图片，使用雪碧图
3. 缓存一切可缓存的资源
4. 使用长Cache
5. 使用外联式引用CSS、JavaScript
6. 压缩HTML、CSS、JavaScript
7. 启用GZip
8. 使用首屏加载
9. 使用按需加载
10. 使用滚屏加载
11. 通过Media Query加载
12. 增加Loading进度条
13. 减少Cookie
14. 避免重定向
15. 异步加载第三方资源



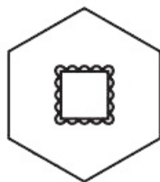
图片优化

1. 使用智图 http://zhitu.tencent.com/
2. 使用（CSS3、SVG、IconFont）代替图片
3. 使用Srcset
4. webP优于JPG
5. PNG8优于GIF
6. 首次加载不大于1014KB（基于3秒联通平均网速所能达到值）
7. 图片不宽于640



脚本优化

1. 减少重绘和回流
2. 缓存Dom选择与计算
3. 缓存列表.length
4. 尽量使用事件代理，避免批量绑定事件
5. 尽量使用ID选择器
6. 使用touchstart、touchend代替click



CSS优化

1. CSS写在头部，JavaScript写在尾部或异步
2. 避免图片和iFrame等的空Src
3. 尽量避免重设图片大小
4. 图片尽量避免使用DataURL
5. 尽量避免写在HTML标签中写Style属性
6. 避免CSS表达式
7. 移除空的CSS规则
8. 正确使用Display的属性
9. 不滥用Float
10. 不滥用Web字体
11. 不声明过多的Font-size
12. 值为0时不需要任何单位
13. 标准化各种浏览器前缀
14. 避免让选择符看起来像正则表达式



渲染优化

1. HTML使用Viewport
2. 减少Dom节点
3. 尽量使用CSS3动画
4. 合理使requestAnimationFrame动画代替setTimeout
5. 适当使用Canvas动画
6. Touchmove、Scroll 事件会导致多次渲染
7. 使用（CSS3 transitions、CSS3 3D transforms、Opacity、Canvas、WebGL、Video）来触发GPU渲染



概述

1. PC优化手段在Mobile侧同样适用
2. 在Mobile侧我们提出三秒种渲染完成首屏指标
3. 基于第二点，首屏加载3秒完成或使用Loading
4. 基于联通3G网络平均338KB/s(2.71Mb/s)，所以首屏资源不应超过1014KB
5. Mobile侧因手机配置原因，除加载外渲染速度也是优化重点
6. 基于第五点，要合理处理代码减少渲染损耗
7. 基于第二、第五点，所有影响首屏加载和渲染的代码应在处理逻辑中后置
8. 加载完成后用户交互使用时也需注意性能

加载优化

加载过程是最为耗时的过程，可能会占到总耗时的80%时间，因此是优化的重点

减少HTTP请求

因为手机浏览器同时响应请求为4个请求（Android支持4个，iOS 5后可支持6个），所以要尽量减少页面的请求数，首次加载同时请求数不能超过4个。

缓存

使用缓存可以减少向服务器的请求数，节省加载时间，所以所有静态资源都要在服务器端设置缓存，并且尽量使用长Cache（长Cache资源的更新可使用时间戳）

- 缓存一切可缓存的资源
- 使用长Cache（使用时间戳更新Cache）
- 使用外联式引用CSS、JavaScript

压缩HTML、CSS、JavaScript

减少资源大小可以加快网页显示速度，所以要对HTML、CSS、JavaScript等进行代码压缩，并在服务器端设置GZip

- 压缩（例如，多余的空格、换行符和缩进）
- 启用GZip

无阻塞

写在HTML头部的JavaScript（无异步），和写在HTML标签中的Style会阻塞页面的渲染，因此CSS放在页面头部并使用Link方式引入，避免在HTML标签中写Style，JavaScript放在页面尾部或使用异步方式加载

使用首屏加载

首屏的快速显示，可以大大提升用户对页面速度的感知，因此应尽量针对首屏的快速显示做优化

按需加载

将不影响首屏的资源和当前屏幕资源不用的资源放到用户需要时才加载，可以大大提升重要资源的显示速度和降低总体流量，但按需加载会导致大量重绘，影响渲染性能。

- LazyLoad
- 滚屏加载

- 通过Media Query加载

预加载

大型重资源页面（如游戏）可使用增加Loading的方法，资源加载完成后再显示页面。但Loading时间过长，会造成用户流失

对用户行为分析，可以在当前页加载下一页资源，提升速度

- 可感知Loading(如进入空间游戏的Loading)
- 不可感知的Loading（如提前加载下一页）

压缩图片

图片是最占流量的资源，因此尽量避免使用他，使用时选择最合适的格式（实现需求的前提下，以大小判断），合适的大小，然后使用智图压缩，同时在代码中用Srcset来按需显示

- 使用智图（ http://zhitu.tencent.com/ ）
- 使用其它方式代替图片(1. 使用CSS3 2. 使用SVG 3. 使用IconFont)
- 使用Srcset
- 选择合适的图片(1. webP优于JPG 2. PNG8优于GIF)
- 选择合适的大小（1. 首次加载不大于1014KB 2. 不宽于640（基于手机屏幕一般宽度））

减少**Cookie**

Cookie会影响加载速度，所以静态资源域名不使用Cookie

避免重定向

重定向会影响加载速度，所以在服务器正确设置避免重定向

异步加载第三方资源

第三方资源不可控会影响页面的加载和显示，因此要异步加载第三方资源

JavaScript优化

脚本处理不当会阻塞页面加载、渲染，因此在使用时需当注意:

- CSS写在头部，JavaScript写在尾部或异步
- 避免图片和iFrame等的空Src: 空Src会重新加载当前页面，影响速度和效率
- 尽量避免重设图片大小: 重设图片大小是指在页面、CSS、JavaScript等中多次重置图片大小，多次重设图片大小会引发图片的多次重绘，影响性能
- 图片尽量避免使用DataURL: DataURL图片没有使用图片的压缩算法文件会变大，并且要解码后再渲染，加载慢耗时长

CSS优化

尽量避免写在**HTML**标签中写**Style**属性

避免**CSS**表达式

CSS表达式的执行需跳出CSS树的渲染，因此请避免CSS表达式

移除空的CSS规则

空的CSS规则增加了CSS文件的大小，且影响CSS树的执行，所以需移除空的CSS规则

正确使用Display的属性

Display属性会影响页面的渲染，因此请合理使用:

a) display:inline后不应该再使用width、height、margin、padding以及float b) display:inline-block后不应该再使用float c) display:block后不应该再使用vertical-align d) display:table-*后不应该再使用margin或者float

不滥用Float

Float在渲染时计算量比较大，尽量减少使用

不滥用Web字体

Web字体需要下载，解析，重绘当前页面，尽量减少使用

不声明过多的Font-size

过多的Font-size引发CSS树的效率

值为0时不需要任何单位

为了浏览器的兼容性和性能，值为0时不要带单位

标准化各种浏览器前缀

- 无前缀应放在最后
- CSS动画只用 （-webkit- 无前缀）两种即可
- 其它前缀为 -webkit- -moz- -ms- 无前缀 四种，（-o-Opera浏览器改用blink内核，所以淘汰）

避免让选择符看起来像正则表达式

高级选择器执行耗时长且不易读懂，避免使用

JavaScript执行优化

减少重绘和回流

- 避免不必要的Dom操作
- 尽量改变Class而不是Style，使用classList代替className
- 避免使用document.write
- 减少drawImage

缓存Dom选择与计算

每次Dom选择都要计算，缓存它

缓存列表.length

每次.length都要计算，用一个变量保存这个值

尽量使用事件代理，避免批量绑定事件

尽量使用ID选择器

TOUCH事件优化

使用touchstart、touchend代替click，因快影响速度快。但应注意Touch响应过快，易引发误操作

渲染优化

HTML使用Viewport

Viewport可以加速页面的渲染，请使用以下代码 `<meta name=”viewport” content=”width=device-width, initial-scale=1">`

减少Dom节点

Dom节点太多影响页面的渲染，应尽量减少Dom节点

动画优化

- 尽量使用CSS3动画
- 合理使用requestAnimationFrame动画代替setTimeout
- 适当使用Canvas动画 5个元素以内使用css动画，5个以上使用Canvas动画（iOS8可使用webGL）

高频事件优化

Touchmove、Scroll 事件可导致多次渲染

- 使用requestAnimationFrame监听帧变化，使得在正确的时间进行渲染
- 增加响应变化的时间间隔，减少重绘次数

GPU加速

CSS中以下属性（CSS3 transitions、CSS3 3D transforms、Opacity、Canvas、WebGL、Video）来触发GPU渲染，请合理使用，过渡使用会引发手机过耗电增加

参考资料

- 移动H5前端性能优化指南
- BlendUI，让webapp的体验和交互得到质的提升

浏览器渲染性能优化

影响性能的因素

- 白屏
    - HTML 和 CSS 的加载及解析速度
    - `<head>` 内的脚本加载及执行
- 首屏
    - 图片加载
    - `<body>` 内的脚本加载及执行
- render tree 的构建
    - HTML 的复杂度
    - CSS 的复杂度
- render tree 的绘制（栅格化）
    - 颜色的复杂度
    - 形状的复杂度

怎么提高前端性能？

提高以下几个方面，总体性能就会得到大幅度提升：

- 缩短白屏时间；
- 加快首屏显示；
- 尽快监听主要操作的事件。

优化关键呈现路径

为了在首次渲染时尽可能快，我们需要优化以下三个变量：

- 最小化关键资源数
- 最小化关键字节数
- 最小化关键路径长度

常规步骤:

- 分析并描述关键路径：资源数、字节数和长度；
- 减少关键资源的数量：删掉、延迟下载或标记为异步等等；
- 优化剩余关键资源的加载顺序：尽早下载所有关键资源以缩短关键路径长度；
- 优化关键字节数以减少下载时间（往返次数）。

PageSpeed 规则和建议

- 排除阻止呈现的 JavaScript 和 CSS
- 优化 JavaScript 的用法
    - 推荐使用异步 JavaScript 资源
    - Avoid synchronous server calls
    - 延迟解析 JavaScript
    - 避免运行时间长的 JavaScript
- 优化 CSS 的用法

- 将 CSS 放到文档头部
- 避免使用 CSS import
- 内联阻止呈现的 CSS

参考资料

- Google: Optimizing Performance

区分开发与部署环境

前端这一块，开发可以很简单的在本地建立几个文件，浏览器就能运行起来。但是实际部署的情况，需要进行页面资源优化。

缓存的使用

本地访问的时候，不管是file还是http的协议，一般不会察觉出加载慢得情况。但是线上的情况却很复杂，每个请求者的网络环境不同，那么导致加载的时间也不一致，为了性能以及带宽。我们会选择使用缓存去减少一些请求，比如不常更新的样式文件。

首先开启缓存机制，需要强制浏览器使用本地缓存。这一块应该是在处理请求的服务器上处理的, 在返回的http headers中标明。

启用了缓存机制，同样的请求默认请求一次，以后刷新请求的都是本地缓存资源。但是如果要更新资源，怎么办？原先我这里的想法是，也是最普遍的想法就是追加时间戳。使得请求路径发生变化，这样就会让浏览器主动放弃缓存，加载新的资源。更好的方法，我看了百度FIS的做法，他们的思路是让url与文件内容关联，使用的是数据摘要算法，对文件求摘要信息，可以精确到单个文件的缓存控制。

普通服务器与CDN的使用

为了进一步提高网站性能，会把静态资源和动态网页分别存放到CDN与普通服务器上。上面说到的那些缓存文件，其实都是存放在CDN上面的，那么问题来了：部署的时候如何替换，动态网页与静态资源不可能同时进行，如果错开更新，肯定会有某一时刻访问出错。

答案就是：灰度发布。用文件的摘要信息来对资源文件进行重命名，把摘要信息放到资源文件发布路径中，这样，内容有修改的资源就变成了一个新的文件发布到线上，不会覆盖已有的资源文件。上线过程中，先全量部署静态资源，再灰度部署页面，整个问题就比较完美的解决了。

Rails

rails通过把静态资源变成erb模板文件，然后加入<%= asset_path 'image.png' %>，上线前预编译完成处理。

参考资料

- 知乎：大公司里怎样开发和部署前端代码？
- Blog:前端农民工
- A Beginner's Guide to HTTP Cache Headers
- RailsGuides: The Asset Pipeline

优化网络请求

如何加快网速把，其实和上一篇应该有些地方重复。具体的资源，见 `assets` 目录。

页面提速方法

减少对服务器的文件请求

常规的HTTP请求属于“请求”-“应答”-“断开”形式的短连接，每一个独立的资源我们都会向服务器发去一份get请求，再等服务端将我们需要的文件传回来。每一次资源的请求都实实在在地耗费了一次“连接-等待-接收”的时间（当然将http请求设为keep-alive长连接状态可以减少“连接”的次数和时间），如果我们能有效减少对服务器文件的请求次数，便意味着我们可以从这块省下一些页面等待时间，也可以顺便减少服务器的负担。

具体的方式可以有：

- 使用css sprite技术合并多个图片为单个图片文件，实际使用时通过background-position来定位背景位置（相信大家第一个想到的也是这个吧）；
- 合并多个css样式文件为单个样式文件，合并多个脚本为单个脚本，再在页面中引用合并后的样式/脚本文件。对于这个你可以使用r.js来帮忙（看我这篇文章），但我个人倒是不怎么推荐这个方法，因为合并了文件之后，多个页面之间公共部分的样式/脚本文件就无法缓存到客户端了；
- 使用base64编码来展示图片。就如图github 404页面那样。常规只推荐你把这种方式使用在用户重复访问量较少的页面，因为它们虽然无须从服务端get一遍，但也无法缓存在客户端，导致用户每次访问页面都要重新渲染一次。而且冗长的文件流代码会占用你页面很大的代码空间，维护起页面来估计也会挺心塞；
- 将小块的css、js代码段直接写在页面上，而非在页面引入独立的样式/脚本文件。相信有的朋友看惯了“保持结构（标记）、表现（样式）、行为（脚本）三者分离”的规范，对此观点可能有些意见。只能说规范不是教条，适合自己的才是硬道理。直接把小段的、复用率低的样式/脚本直接写于页面上带来的利还是大于弊的（弊可能也就是增大了页面代码量、不那么好维护了点）。反观所有主流门户网站的页面源文件，基本没有一个是把样式/脚本都全部作为外部文件引入的（无论他们是否从减少服务器请求这点出发，事实都是这样）；
- 利用http-equiv="expires"元标签，设定一个未来的某时间点作为页面文件过期时间，用户在过期时间之前所获取到的页面文件都仅从缓存中去取。不过这个办法太死板（有时候即使服务端及时把过期时间更改为已结束时间，客户端可能都不会按照新更改的规则去服务端获取新文件资源），常规是不推荐使用的。

减少文件大小

文件太大（特别是图片）导致加载时间较长，往往都是影响页面加载体验的头号大敌，那么尽可能减少请求文件的大小便是相当重要的事情了，我们可以做的事情有：

- 压缩样式/脚本文件，就此你可以使用gulp或者grunt来实现这点，它们均能很好地减少css/js文件的大小（对于js还能起到混淆变量、函数名的作用）；
- 针对性选择图片格式，在无透明背景需求下，对于颜色较单一、无色彩渐变的图片仅使用gif格式，对于jpg图片也可按照其清晰度要求，在导出jpg的时候选择对应的“品质”进行优化。如果你喜欢尝鲜，可以学淘宝那样使用webp图片格式，它能很好地优化同画质下的文件大小。
- 使用Font Awesome来替代页面上的图标，其原理是使用@font-face让用户下载一个非常小的UI字体包，把页面上用到的图标以字符的形式来显示，从而减少了图片需求和图标文件大小。

适度使用CDN

使用CDN有几个好处：如果用户在其它站点下载过这个CDN资源，那么来我们站点仅仅从缓存获取即可；减少了对自己站点服务器的文件请求（外部CDN的情况下），减少服务器负担；多个域会使浏览器允许异步下载资源的最大数量增多，比如一个站点只从一个域来请求资源，那么FireFox只允许同时刻最多异步下载2个文件，但如果使用了外部CDN来引入资源，那么FF允许在同时异步下载本域中的两个资源外，还额外允许同时异步下载另一个域（CDN）下的2个资源。

但是使用CDN有一个很大的问题——增加了dns解析的开销，如果一个页面同时引入了多个CDN的资源，可能会因为dns解析

而陷入较多的等待时间，导致得不偿失。

对于这个问题，常规是建议一个站点下只使用同一个可靠、快速的CDN来引入各种所需资源即可，也就是说，建议一个页面从2个不同的域（比如站点域和CDN域）下来请求资源是最佳的选择（据说这个结论是雅虎前端工程师提出的，这里有一篇很不错的文章）。

延迟请求、异步加载脚本

在各主流浏览器下，常规情况，我们的脚本文件跟随其它资源文件一样都是异步下载的，但这里存在一个问题——比如FireFox下载好脚本后的一小段时间内会有"执行阻塞"的情况发生，也就是说浏览器下载好脚本后执行它的这段时间里，浏览器的其它行为被阻塞，导致页面上的其它资源都是无法被请求和下载的.

如果你页面里存在js代码执行时间过长的情况，那么用户就会明显感觉到页面的延迟。解决这个问题有一个简单的方法——将脚本请求标签放置到结束标签前，使得页面上的脚本成为最后被请求的资源，自然也不会阻塞其它页面资源的请求事件了。

另外，虽然上面提到"我们的脚本文件跟随其它资源文件一样都是异步下载的"，但异步下载不代表异步执行，为了严格保证脚本逻辑顺序和依赖关系的正确性，浏览器会按照脚本被请求的先后顺序来执行脚本。那么问题就来了——如果页面上的脚本依赖关系并不大，甚至没有任何相互间的依赖，那么浏览器的这套规则就仅仅增加了页面请求阻塞时间而已（就像你花大钱买了一笔保险，但被保险期间你平安无事啥都没发生。。。嗯，这个比喻有点反人类。。。）。

解决这个问题的办法无非就是让脚本无阻塞地异步执行，比如给script标签加上defer和async属性或者动态注入脚本（可以参考这里），但这些都不是良好的解决方案，要么存在兼容性问题，要么太麻烦还无法处理依赖。

个人是推荐使用 requireJS（AMD规范） 或 seaJS（CMD规范） 来异步加载脚本并处理模块依赖的，前者将"依赖前置"（预加载所有被依赖脚本模块，执行速度最快），后者走的"依赖就近"（懒加载被依赖脚本模块，请求脚本更科学），你可以根据项目具体需求来选择最合适的。

延迟请求首屏外的文件

先解释下，"首屏"指的是页面初始化时候的页面内容显示区域，也就是页面一加载，用户就首先看到的区域。

比如像京东啊淘宝啊，对于需要滚动页面才能看到的图片内容，都做了类似lazyload的处理，这些无非都是走了代理模式的理念，但的确给用户一个错觉——这个页面更快地加载完了，因为我很快就看到了屏幕上的内容（即使我还没下拉滚动条，而页面后方的文件其实还没真正加载呢）。

我们可以这样实现此方案，不依赖任何lazyload库，拿图片来做示范，我们可以这样编写首屏外的图片（假设某张图片地址是a.jpg）的img标签：

```
<img src="data:image/png;base64,R0lGODlhAQABAAD/ACwAAAAAAQABAAACADs=" data-src="a.jpg">
```

如上所示，页面初步加载这张图片的时候是直接以base64的方式（当然你也可以统一使用一张占位图loading.gif来替代）来快速显示一张极小的图片的，而图片本身的真实路径是存在data-src属性内的，我们可以在页面加载结束后再向服务器请求它真实的文件并替换：

```
function init() {
    var imgDefer = document.getElementsByTagName('img');
    for (var i=0; i<imgDefer.length; i++) {
        if(imgDefer[i].getAttribute('data-src')) {
             imgDefer[i].setAttribute('src',imgDefer[i].getAttribute('data-src'));
        }
    }
}
window.onload = init;
```

```
```

如上是对图片的延迟加载处理，对于视频、音频文件，可以采取完全一样的原理来延迟加载，从而有效减少页面初始化等待时间。

优化页面模块排放顺序

这里有一个很好的例子，比如有一个页面是这样的——左边是侧边栏，用于存放用户的头像啊、资料啊，以及网站投放的广告啊，而右侧是文章内容区域：

```
<body>
  <sidebar>
    <!-- 侧边栏内容 -->
  </sidebar>

  <content>
    <!-- 文章内容 -->
  </content>
</body>
```

于是乎，浏览器按照它的UI单线程准则从上到下先加载了侧边栏，再加载我们的文章。。。

很明显，这样不是一个人性化的加载顺序，我们得弄清楚，页面上各个区域模块，对于用户而言，哪个才是最重要、最应当首先展示的。

对于上面的例子，文章内容才应该是用户首先要看到、需要浏览器优先请求和显示的区域。所以我们得修改我们的代码为：

```
<body>
  <content>
    <!-- 文章内容 -->
  </content>

  <sidebar>
    <!-- 侧边栏内容 -->
  </sidebar>
</body>
```

当然这里仅仅是用一个小示例来挑起各位的脑洞，懂得举一反三和实际运用才是硬道理。

其它建议

1. 不要在css中使用@import，它会让一个样式文件去等待另一个样式文件的请求，无形中增加了页面等待时间（当然如果走的scss，@import就是另一回事了，呃跑题了~）；
2. 避免页面或者页面文件重定向查找，这相当于你走进了一间卫生间，然后看到上面的牌子说“此处不同，请去前面左拐的卫生间”，又得重走一遍；
3. 减少无效请求——比如通过css/js来请求一个不存在的资源，可能会导致较长的等待和阻塞（直到它返回错误信息）；
4. 无论你是否决定将脚本放到页尾，但一定要保障脚本放置于样式文件后方；
5. 文件在小于50K的时候，直接读取文件流会比从文件系统中去读取文件来的快些，大于50K则相反。比如有一张图片，如果它小于50K，我们可以将它转为二进制数据存储在数据库中，页面若要读取该图片则从数据库上来读取，若文件大小大于50K，那建议存放在可访问的文件夹中以文件的形式来读取即可；
6. 使用 cookie-free domains 来存放资源，减少无用cookie传输的网络开销（可以参考这篇文章）；
7. 配置.htaccess文件、走Gzip页面压缩形式、开启keep-alive连接模式等后端解决方案，这边就不细说了。

参考资料

- 浅谈WEB页面提速（前端向）
- Font Awesome
- 探真无阻塞加载javascript脚本技术，我们会发现很多意想不到的秘密
- 优设：浓缩的精华！从零开始带你认识最新的图片格式WEBP

Chrome开发者工具的使用

参考资料

- Chrome DevTools
- Chrome 控制台不完全指南
- Chrome控制台 如何调试Javascript
- Chrome 控制台console的用法
- Chrome开发者工具之JavaScript内存分析
- 译：使用Chrome开发工具调试Canvas

JavaScript内存优化

参考资料

- Chrome开发者工具之JavaScript内存分析
- 了解 JavaScript 应用程序中的内存泄漏
- JavaScript中的内存泄露模式
- 案例分析之JavaScript代码优化

javascript事件优化

javascript是如何切入到html和css中间，让三者融合呢？最后我发现这个切入点就是javascript的事件系统，不管我们写多长多复杂的javascript代码，最终都是通过事件系统体现在html和css上，因此我就在想既然事件系统是三者融合的切入点，那么一个页面里，特别是当今越来越复杂的网页里必然会有大量事件操作，没有这些事件我们精心编写的javascript代码只有刀枪入库，英雄无用武之地了。

HTML事件处理

html事件处理就是将事件函数直接写在html标签里，因为这种写法和html标签紧耦合，所以称为html事件处理。例如下面代码：

```
<input type="button" id="btn" name="btn" onclick="alert('Click Me!')"/>
```

or

```
<input type="button" id="btn" name="btn" onclick="btnClk()"/>

function btnClk(){

        alert("click me!");

}
```

上面这个写法是一种很美的写法，所以时下还是很多人会不自觉的使用它，但是也许很多人不知道，后一种写法其实没有前一种写法健壮，这个也是我前不久在研究非阻塞加载脚本技术时候碰到的问题，因为根据前端优化的原则，javascript代码往往是位于页面的底部，当页面有被脚本阻塞时候，html标签里引用的函数可能还没执行到，这个时候我们点击页面按钮，结果会报出“XXX函数未定义的错误”，在javascript里这样的错误是会被try，catch所捕获，因此为了让代码更加健壮，我们会有如下的改写：

```
<input type="button" id="btn" name="btn" onclick="try{btnClk();}catch(e){}"/>
```

但是这是一种极其不推荐的做法！耦合性太强！

DOM0级事件处理

DOM0级事件处理是当今所有浏览器都支持的事件处理，不存在任何兼容性问题，看到这样一句话都会让每个做web前端的人们激动不已。DOM0事件处理的规则是：每个DOM元素都有自己的事件处理属性，该属性可以赋值一个函数，例如下面的代码：

```
var btnDOM = document.getElementById("btn");

btnDOM.onclick = function(){

        alert("click me!");

}
```

DOM0级事件处理的事件属性都是采用“on+事件名称”的方式定义，整个属性都是小写字母。这个事件处理的方式只能绑定一

个函数，多个函数的话，后面一个函数会将之前的函数覆盖。

DOM2事件处理和IE事件处理

DOM2事件处理是标准化的事件处理方案，但是IE浏览器自己搞了一套，功能和DOM2事件处理相似，但是代码写起来就不太一样了。DOM2事件处理在ie9包括ie9以上的版本都得到了很好的支持，ie8以下是不支持DOM2事件的。旧版本的IE使用的是自己封装的一套 `attachEvent` ,标准的是使用 `addEventListener` 。

事件流

在页面开发里我们常常会碰到这样的情况，一个页面的工作区间在javascript可以用document表示，页面里有个div，div等于是覆盖在document元素上，div里面有个button元素，button元素是覆盖在div上，也等于覆盖着document上，所以问题来了，当我们点击这个按钮时候，这个点击行为其实不仅仅发生在button之上，div和document都被作用了点击操作，按逻辑这三个元素都是可以促发点击事件的，而事件流正是描述上述场景的概念，事件流的意思是：从页面接收事件的顺序。

事件冒泡和事件捕获

- 事件冒泡: 是微软公司提出解决事件流问题的方案
- 事件捕获: 是网景公司提出的事件流解决方案

冒泡事件由div开始，其次是body，最后是document，事件捕获则是倒过来的先是document，其次是body，最后是目标元素div，相比之下，微软公司的方案更加人性化符合人们的操作习惯，网景的方案就很别扭了，这是浏览器大战的恶果。

attachEvent

微软为自己的事件方式选择了一套做法： `attachEvent` ,在ie下通过DOM元素的attachEvent方法添加事件，和DOM0事件处理相比，添加事件的方式由属性变成了方法，所以我们添加事件就需要往方法里传递参数，attachEvent方法接收两个参数，第一个参数是事件类型，事件类型的命名和DOM0事件处理里的事件命名一样，第二个参数是事件函数了，使用方法的好处就是如果我们在为同一个元素添加个点击事件。

但是需要注意的是删除事件的时候，如果绑定传入的是匿名函数，是无法被清除干净的。因此写事件要有个良好的习惯即操作函数要独立定义，不要用匿名函数用成了习惯。

addEventListener

DOM2是标准化的事件，使用DOM2事件，事件传递首先从捕获方式开始即从document开始，再到body，div是一个中介点，事件到了中介点时候事件就处于目标阶段，事件进入目标阶段后事件就开始冒泡处理方式，最后事件在document上结束。（捕获事件的起点以及冒泡事件的终点，我本文都是指向document，实际情况是有些浏览器会从window开始捕获，window结束冒泡，不过我觉得开发时候不管浏览器本身怎么设定，我们关注document更具开发意义，所以我这里一律都是使用document）。人们习惯把目标阶段归为冒泡的一部分，这主要是因为开发里冒泡事件使用的更加广泛。

DOM2事件处理里添加事件使用的是addEventListener，它接收三个参数比ie事件处理多一个，前两个的意思和ie事件处理方法的两个参数一样，唯一的区别就是第一个参数里要去掉on这个前缀，第三个参数是个布尔值，默认为**false**。如果它的取值是true，那么事件就按照捕获方式处理，取值为false，事件就是按照冒泡处理，有第三个参数我们可以理解为什么DOM2事件处理里要把事件元素跑个两遍，目的就是为了兼容两种事件模型，不过这里要请注意下，不管我们选择是捕获还是冒泡，两遍遍历是永远进行，如果我们选择一种事件处理方式，那么另外一个事件处理流程里就不会促发任何事件处理函数，这和汽车挂空挡空转的道理一样。通过DOM2事件方法的设计，我们知道DOM2事件在运行时候只能执行两种事件处理方式中的一种，不可能两个事件流体系同时促发，所以虽然元素遍历两遍，但是事件函数绝不可能被促发两遍。

三种方式的比较

HTML事件处理是绝对不推荐的方式，DOM0事件一个DOM元素某个事件有且只有一次，DOM2事件可以让DOM元素某个事

件拥有多个事件处理函数，且能让我们精确控制事件流的方式。

性能优化

两个着力点来思考事件系统的性能问题，它们分别是：

- 减少遍历次数
- 内存消耗

遍历次数

不管是捕获事件流还是冒泡事件流，都会遍历元素，而是都是从最上层的window或document开始的遍历，假如页面DOM元素父子关系很深，那么遍历的元素越多，像DOM2事件处理这种，遍历危害程度就越大了，如何解决这个事件流遍历问题了？我的回答是没有，这里有些朋友也许会有疑问，怎么会没有了？事件系统里有个事件对象即event，这个对象有阻止冒泡或捕获事件的方法，我怎么说没有呢？这位朋友的疑问很有道理，但是如果我们要使用该方法减少遍历，那么我们代码就要处理父子元素的关系，爷孙元素关系，如果页面元素嵌套很多，这就是没法完成的任务，所以我的回答是没法改变遍历的问题，只能去适应它。

内存消耗

在javascript里，每个函数都是一个对象，每个对象都会耗费内存。在当今ajax流行，单页面开发疯狂普及的时代，一个网页上的事件都是超级多的，这就意味我们每个事件都有一个事件函数，但是我们每次操作都只会促发一个事件，此时其他事件都是躺着睡觉，起不到任何作用同时还要消耗计算机的内存。

在讲述DOM2事件处理里我提到了目标对象这个概念，抛开DOM2事件处理方式，在捕获事件处理和冒泡事件处理里也有目标对象的概念，目标对象就是事件具体操作的DOM元素，例如点击按钮操作里按钮就是目标对象，不管哪个事件处理方式，事件函数都会包含一个event对象，event对象有个属性target，**target**是永远指向目标对象的，event对象还有个属性就是currentTarget，**currentTarget**指向的是捕获或冒泡事件流动到的**DOM**元素。

假如我们在document上添加点击事件，页面上的按钮不添加点击事件，这时候我们点击按钮，我们知道document上的点击事件会促发，这里有个细节就是促发document点击事件时候，event的target的指向是button而不是document。

事件委托

使用事件委托时可以避免问题的发生，例如将事件绑定在document，document代表整个页面，所以它加载完毕的时间可谓最早，所以在document上实现事件委托，就很难发生事件无效的情况，也很难发生浏览器报出“XXX函数未定义”的问题了。总结一下这个特点：事件委托代码可以运行在页面加载的任何阶段，这点对提升网页性能还是增强网页效果上都会给开发人员提供更大自由度。

参考资料

- 关于编写性能高效的javascript事件的技术

页面滚动性能

参考资料

- Javascript高性能动画与页面渲染
- 页面滚动性能
- pointer-events:none提高页面滚动时候的绘制性能？

web开发中的坑

开这一章节，记录一下自己开发过程中遇到的问题，都是些通用的问题，作为经验总结。

排版兼容性问题

IE下的问题

IE8

- 不支持css3属性，特别是背景图片size,position,对于圆角可使用PIE.js
- 无法使用media query, 通过插件完成
- `<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">` 强制IE8使用最新的内核渲染页面
- 不支持图片opacity,但是可以设置滤镜完成
- max-width在嵌套下有问题

IE9

- canvas的height不能设置为auto,要么固定，要么动态计算赋值,建议使用Img替代canvas

Firefox

- 图片动画抖动，添加translate3d(0,0,0)

Chrome

通用

- 如何解决inline-block元素的空白间距

参考资料

- 前端兼容性不完全指南
- 「HTML5,CSS3」viewport 兼容性问题
- 网页布局中的常见的兼容性问题

浏览器报错

记录下遇到的浏览器报错。

iPad的bug合集

ipad上滚动条，在滚动过程之中，无法捕获滚动高度？

双指滑动的过程之中，js不会被触发。除非全部接管touch事件，比如使用skrollr.

- How can I monitor scroll position while scrolling in Safari on iOS?
- Get scrollTop() while scrolling on iPad/iPhone

fixed布局元素宽度在旋转的时候会有计算问题

总体来说，在safari下尽量少使用fixed布局。

参考资料

localStorage的使用

这是一个HTML5提供的本地存储方案，它会在本地存储一个key,value的对象。

添加标识

如果需要使用localStorage存储数据的话，主要为每个值添加唯一的标识。不然很容易读取到其他程序存储的内容。

隐私模式下注意点

PC端,这两个API在低版本的IE下是没有,所以是需要用try..catch包裹的.

在移动端,我刚刚开始是不加的,所测试的手机也没问题.但是现在很多浏览器有无痕模式,这个模式下,localStorage相关的API时禁用的.所以使用时,还是要保证代码的健壮性， 添加一个polyfill.

```
// 解决隐私模式下 localStorage 不正常问题
;(function() {
  var KEY = '_localStorage_'
    , VALUE = 'test';

  // 检测是否正常
  try {
    localStorage.setItem(KEY, VALUE);
  } catch(e) {
    var noop = function() {};

    localStorage.__proto__ = {
      setItem: noop,
      getItem: noop,
      removeItem: noop,
      clear: noop
    };
  }

  // 删除测试数据
  if(localStorage.getItem(KEY) === VALUE) localStorage.removeItem(KEY);
})();
```

ios7下另一个问题

上面的这段代码，有一个严重的隐患，就是在原型的 `__proto__` 属性上添加东西，这一点不是所有浏览器都支持的，所以我在ios7下会得到报错。

解决方案：建议使用其他命名替代 `localstorage` 或者针对 `localstorage` 部分使用try,catch包装。

移动端fixed布局

在移动开发的过程中，元素使用fixed布局在safari下经常会遇到问题，比如：

- ios6下，不固定，滑动会突然跳动
- 输入框在弹出键盘后，不吸附在键盘上方
- fixed的元素宽度不能自适应，在ios6下横竖屏切换的过程中
- iframe下，fixed元素高度超大

参考资料

- 移动Web开发，4行代码检测浏览器是否支持position:fixed
- 移动端web页面使用position:fixed问题总结
- 移动Web开发实践——解决position:fixed自适应BUG

touch事件

移动开发过程之中，涉及到touch事件的情况特别多，比如按下，按起，滑动。这些事件在不同的设备之中存在差异。

组件

之前的滑动处理我都是自己去写的，没有使用第三方的组件。看了一篇文章介绍，提到了三个组件：

- hammer.js
- zepto
- appframework

这三个都对touch进行了处理，第一个是专门的触屏手势处理框架，功能很强大。其他两个，据说zepto的会好一点，虽然appframework自己说是对zepto的touch进行了修改和提升。

事件顺序

没有touch的时候: mousedown -> mouseup -> click。

有touch的时候：touchstart -> touchmove -> touchend。

都有的时候:

- 点击操作: touchstart -> touchend -> mousedown -> mouseup -> click
- 移动的时候: touchstart -> touchmove -> touchend

坑

安卓设备下 `swipe` 很难触发

这是安卓的一个老问题了.谷歌一下 类似zepto android swipe的关键字,就能发现不少.很多项目中的issue都能找到这个问题,连安卓自身项目里都有相关issue(链接要翻墙)。

解决方法就是在touchstart或touchmove事件中,主动调用e.preventDefault()。

滑动方向的问题

以上下滑动为主的操作,有可能触发swipeLeft/Right事件,而不是swipeUp/Down事件.这个主要是因为native scroll的一些特性,导致算距离时竖向的距离可能会很小,而上下滑的最后,一般会有一个横向滑动的连带动作,导致deltaX>deltaY,从而判断出错.

最后说一下,还是慎用左右滑动的设计,一是兼容性问题.二是,微信等app是手机网页主要的入口,而从左向右的滑动,很容易关闭页面.

参考资料

- 记mobile web曾经的踩过坑

触摸和鼠标在一起

简介

近三十年，桌面计算体验集中围绕在一个键盘与一个鼠标或轨迹板，以它们作为主要的用户输入设备。不过在过去十年间，智能手机和平板电脑带来了一个新的交互范式：触摸。通过引入触屏的Windows 8机器，以及现今很酷的触摸式Chromebook Pixel笔记本的发布，触摸现已成为桌面体验的部分预期。最大的挑战之一是创造出不仅仅工作于触摸设备和鼠标设备的体验，还要创造出用户同时使用这两种输入方法的体验——有时候是同时的！

这篇文章将帮助你理解触摸功能是如何内置于浏览器的，怎样将这种新的界面机制集成到你已有的应用，以及触摸怎样才可以和鼠标输入和谐共存。

Web平台中的触摸状态

iPhone是第一个在web浏览器中植入了专用于触摸的API的流行平台。有些其它的浏览器制造方创建了类似的API接口，植入浏览器并与iOS的实现兼容，现在被描述为"触摸事件版本1"规范。触摸事件在桌面环境被Chrome和Firefox支持，在iOS环境被Safari支持，在Android环境被Android浏览器支持，还有其它的移动浏览器比如Blackberry浏览器。

我的同事Boris Smus写了一篇关于触摸事件的HTML5Rocks教程，如果你以前没有看过触摸事件，这仍然是一个很好的开始。事实上，如果你以前没有处理过触摸事件，在你继续以前，现在就去阅读那篇文章吧。你去吧，我等着。

结束了？现在你对触摸事件有了一些基础知识，写一个可触摸的交互的挑战在于，触摸交互与鼠标（以及仿真鼠标的轨迹板和轨迹球）事件极为不同——虽然典型的触摸接口试图在模仿鼠标，但这种模仿并不完美或者说完整；你确实需要处理这两种交互模式，而且可能会不得不单独的支持每一个接口。

用户或许有触屏和一个鼠标

许多开发人员创建的网站会静态的检测环境是否支持触摸事件，之后再假设他们只需支持触摸（不需要支持鼠标）事件。现在这是一个错误的假设**——**相反的，仅仅因为触摸事件的存在并不意味着用户主要使用的就是触摸输入设备。像Chromebook Pixel笔记本之类的设备以及一些Windows 8便携电脑现在已经可以同时支持鼠标与触摸式输入方法，在不久的未来还会有更多。在这些设备中，用户同时使用鼠标和触屏与应用交互是很自然的事情，所以“支持触屏”并不等于“不需要支持鼠标”。你不能将这个问题想成“我需要写两种不同的交互方式并且在它们之间切换”，你要想清楚这两种交互怎样独立的工作，也要想清楚怎样让它们协同工作。在我的Chromebook Pixel笔记本上，我经常使用轨迹板，但也会伸手触摸屏幕——在同一个应用或页面中，当时怎么感觉自然我就怎么做。从另一方面来说，有些触屏的便携式电脑用户几乎从不使用触屏——因此触摸输入的存在不应该禁止或者隐藏鼠标控制。

不幸的是, 很难知道用户的浏览器环境是否支持触摸输入,理想情况下,台式机上的浏览器应该显示对触摸事件的支持，这样就可以随时安装触摸屏显示器（例如：通过KVM连接触摸屏是否可行）. 基于所有这些原因，您的应用不应该尝试在触摸和鼠标之间切换**--**只需要两种方式都支持**.**

指针事件

在Windows8系统的IE10浏览器中. 微软引入了一种叫做指针事件的心的模型.指针事件是鼠标事件和触摸输入事件，还有其他输入方式比如笔输入的联合. 将指针事件模型提交到W3C标准还有很多工作要做,在短时间内, 已经有像PointerEvents 和Hand.js 这样的类库供你在代码中实现指针事件，从而避免为鼠标和触摸分别提供支持. 为了更好的触摸和鼠标交互,你可能需要为鼠标和触摸事件分别自定义用户体验,但在很多情况下,统一事件处理都简化了这样的过程. 然而,这种模型还面临着巨大的挑战,它需要支持冗余的输入模型,也还没有被广泛支持,而且还需要很多事件来将它变成一个稳定的,跨浏览器的标准.

与此同时,最好的建议是同时支持鼠标和触摸交互模型.同时支持触摸和鼠标事件还面临着很多挑战,所以这篇文章对这些挑战以及克服这些挑战的策略进行了分析。另外,有些建议只是一般的“实现触摸”的建议,所以，如果你已经熟悉了在移动环境中实现触摸,这可能是多余的.

同时支持鼠标和触屏

点击和轻拍-“自然的”事物的顺序

第一个问题是传统的触摸界面技术想要模仿鼠标的点击-很显然，在触屏技术应用到应用程序之前，是仅仅只能和鼠标事件进行交互的。你可以把这个作为一个快捷键使用-因为“点击”事件将会继续被淘汰，无论用户是用鼠标点击还是用手指轻敲屏幕。然而，这个快捷方式还有一些问题。

首先，你在设计先进的触屏交互技术的时候必须要很仔细。：当用户使用鼠标，它就会通过点击事件给出应答，但是当用户触摸屏幕的时候，触摸和点击事件都会发生。对于一个简单的点击事件，其顺序是:

1. 触屏开始
2. 触屏移动
3. 触屏结束
4. 鼠标悬停
5. 鼠标移动
6. 鼠标按下
7. 鼠标弹起
8. 点击

当然，这个也意味着如果你正在处理触屏事件，比如说触屏开始，你需要确定你没有在同时处理相应的鼠标按下以及/或是点击事件。如果你能取消这个触屏事件（在事件处理程序中访问perventDefault()方法）,然后在这次触屏中没有鼠标事件出现。其中触摸处理程序最重要的一个规则是: 使用事件处理程序中的**perventDefault()**方法，所以默认的鼠标仿真处理就不会发生。.

然而，这样也限制了其他默认浏览器的的行为（像scrolling）-虽然通常你在你的处理程序中完全的处理触屏事件，并且你想要禁止默认的行为。一般来说，你要么去处理和取消所有触屏事件，要么避免有一个对应这个事件的处理程序。

其次，当用户在移动设备上触摸一个网页上的某个元素时，相对于鼠标事件(mousedown)处理，那些没有为移动设备交互做专门设计的网页处理**touchstart**事件会有一个至少**300**毫秒的延迟。若你身边有触控设备，可以试试这个 example，看看这个延迟效果。或者，也可以使用Chrome，打开Chrome开发者工具中打开 "Emulate touch events"，可以帮助你在非触控系统上测试触控接口。

这个延迟是用来给浏览器判定用户是否在采用其他的手势操作，特别是双点缩放。很明显，这个延迟在需要对手指点击做出瞬时相应时会引起问题。已经有个正在进行中的工作尝试对这些会由于延迟而引起问题的场景做出限制。

鼠标移动事件不是通过触摸实现的

在这一点上,非常值得注意的是,通过触摸接口对于鼠标事件的仿真通常并不扩展到仿真鼠标移动事件,所以如果你创建了一个使用鼠标移动事件的鼠标驱动控制,它可能不会再可触摸设备上很好的工作,除非你也明确的为其添加触摸移动事件的处理程序.

浏览器在Html控制上通常会自动实现对于触摸交互的适当响应 - 所以,比如,Html5Range控制只会在你使用触摸交互的时候起作用. 然而,如果你已经实现了自己的控制,它们可能不会响应点击拖动类型的交互;实际上,一些通用的类库(比如jQueryUI)还没有像这样实现本地化的支持触摸交互（尽管jQueryUI提供了像“猴子补丁”一样的修补来解决这个问题）. 这是我在升级我的Web Audio Playgroud应用来支持触摸事件时遇到的第一个问题 - 滑动条是基于jQueryUI的,所以它们不支持点击拖动交互. 我切换成 HTML5 Range 控制, 然后就可以了.当然,我简单地添加了触摸移动处理程序来升级滑动条,但是还有一个问题...

触摸移动和鼠标移动不是同一件事

我曾经见过一些开发者陷入的一个误区就是：让触摸移动和鼠标移动的处理器调用相同的代码. 这些事件的行为非常相近,但存在细微的不同- 特别是, 触摸事件总是以触摸发生时所在的元素为目标，然而鼠标事件则以当前位于鼠标指针下方的元素为目标. 这就是为什么我们有鼠标移上和鼠标移出事件，却没有响应的触摸移上和触摸移出事件,只有触摸事件.

对此最常见的刺痛你的方式就是如果你碰巧移除（或者重新定位）某个用户刚开始触摸的元素. 比如,假设一个图像切换模块中有一个触摸处理器来支持特定的滚动行为.随着现有图片的变更,你移除了一些元素，并添加了新的元素.如果用户碰巧开始触摸在其中的一张图片上,然后你又移除了它，你的处理器（作用于图片元素的祖先元素）将停止接收触摸事件（因为它们被分配到了一个当前DOM树中不存在的目标上）-那么看起来将是用户正把手指放在一个位置，尽管这个位置的元素已经移动了并且最后被移除了.

当然，你可以通过避免移除触摸事件开始时已经绑定触摸处理器的元素（或者祖先元素已经绑定处理器的元素）. 或者,最好的方式是先不注册touchend或者touchmove处理器，一直等到获得了touchstart事件，然后为touchstart事件的目标元素添加touchmove/touchend/touchcancel处理器（然后在end/cancel事件发生时移除这些处理器）.这样你就可以一直接收触摸事件，即使目标元素移动了或者被移除了.你可以在这里尝试-触摸红色的方框然后点击将它从DOM中移除.

触摸和 :Hover

鼠标指针把光标位置和动态选择区分开来,这使得开发者可以使用"移上"状态来隐藏和显示跟用户相关的信息.然而，很多触摸接口当前都不能检测到手指"悬浮"在某个目标上面- 所以，通过这种方式提供重要的语义信息（比如，提供"这个控制是什么?"弹出层）是不可行的, 除非你提供了一种触摸友好的方式来提供信息. 对于如何使用hovering来向用户展示信息你需要谨慎.

然而，足够有趣的是, CSS的:hover伪类在某些情况下可以通过触摸触发- 轻巧某个元素使其具有：hover状态时当手指按下的时候，它也获得了:hover状态.(在IE中,:hover只有当用户的手指按下的时候才起作用,其他浏览器中:hover被一直保持有效直到下一次敲打或者鼠标移动)这是一种在触摸接口中实现弹出菜单的有效方式-副作用就在于此时:hover状态也触发了.例如:

```
<style>
img ~ .content {
  display:none;
}

img:hover ~ .content {
  display:block;
}
</style>

<img src="/awesome.png">
<div class="content">This is an awesome picture of me</div>
```

一旦另一个元素被敲击，当前元素就不在处于活跃状态，hover状态也会消失，就像用户使用鼠标时将鼠标指针移到了元素外面一样. 你可能也希望把内容包裹在一个 `<a>` 元素中来实现制表位的效果，那样用户可以通过鼠标移上或者点击，触摸敲击或者键盘按下来显示或者隐藏额外信息，而不需要Javascript控制.当我开始让我的Web Audio Playground 能够使用触摸接口时，弹出菜单已经能够很好的响应触摸事件的时候我很高兴，因为我已经用过这种结构了!

上面的方式在基于鼠标指针的接口和触摸接口中都能够很好的工作.这与"title"是相比较而言的，它在元素被激活的时候将不会显示出来:

```
<img src="/awesome.png" title="this doesn’t show up in touch">
```

触摸精度 vs. 鼠标精度

鼠标跟现实中的老鼠在概念上是分离的,区别在于他们非常精确,因为底层操作系统通常会追踪指针的精确像素精度. 移动开发者另一方面已经知道手指在触摸屏上的触摸并不如此准确,主要是由于跟屏幕交互时手指表面积的尺寸太大（部分原因是由于你的手指挡住了屏幕）.

很多个人和公司都对如何设计能够容纳基于手指交互的应用和网站进行了大量的用户研究,很多书也是关于这个话题的.基本的意见就是通过增加填补空间(padding)来增加目标对象的尺寸,然后通过增加元素之间的间距(margin)来降低错误敲击的可能性.(Margins不包含在处理触摸和点击事件的敲击检测中,padding却包含在这其中)对于 Web Audio Playground系统一个主要的修补工作就是增加连接点的尺寸，这样他们就能够更准确的被触摸.

很多浏览器厂商在处理基于触摸接口的时候也进入了逻辑来帮助在用户触摸屏幕时准确定位目标元素，同时降错误点击的可能性-尽管这样做通常修正的是点击事件，而不是移动事件（尽管IE好像也修改了mousedown/mousemove/mouseup事件）.

有限的使用触控处理器, 否则滚屏会卡顿

把你的触控处理器限制在你需要他们的地方也很重要;触控元素可能非常消耗带宽，所以滚动屏幕的时候要尽量避免引发触控处理器（因为你的触控处理器可能会干扰到浏览器滚屏优化-现代的浏览器会试着用显卡线程来处理屏幕滚动，但如果他们每次都要检查javascript来知道是否有事件需要被app处理，那基本上这个优化就废了。). 你可以试试这里的浏览器行为的一个示例.

一条避免这个问题的技巧就是只在你的ui中很小一部分使用触控事件处理器, 把你的触控处理器放在这里(比如，不要放在整个页面的body标签里。); 简而言之，尽可能的限制你的触控处理器的使用范围.

多点触控

最后一件有趣而又具有挑战性的事是，尽管我们称之为“触控”用户界面，几乎所有的支持都是多点触控的，也就是说应用程序接口支持每次不止一次触控输入。当你准备在你的应用中支持触控时，你应该考虑多点触控会如何影响你的应用程序。

如果你开发了主要靠鼠标驱动的程序的话，那么你会习惯于用至多一个光标点的系统建立，它不会典型地支持多个光标。对于大多数应用，你将会仅仅把触控事件映射到一个单独的光标接口，但是，我们见过的大多数桌面触控输入软件可以处理至少2次同时的输入，而且大多数新的软件似乎支持至少5次同时的输入。比如说开发屏幕钢琴按键，你要能够支持同时的多点触控输入。 目前实现了的W3C触控API接口没有能决定软件能够支持多少触控点的API，因此你将不得不尽最大努力估计你的用户将需要多少触控点，或者注意观察现实中需要多少触控点并适应之。例如，在一个钢琴应用中，如果你从没见过需要两个以上的触控点，那么你可能会想增加一些“和弦”界面了。PointerEvents API就有一个能够决定“和弦”性能的接口。

触摸起来

我希望这篇文章给你提供了一些指导，有关于实现触摸与鼠标交互时遇到的普遍难题。当然，比任何其他建议都重要的是，你应该在手机，平板电脑，还有混合了鼠标和触摸的桌面环境下测试你的应用。如果你没有触摸与鼠标硬件，可以使用Chrome的"模拟触摸事件"，以便测试不同的场景。

依据这些指导创建吸引人的互动体验，使其良好的工作于触摸输入，鼠标输入，还有甚至是同时这两种交互，这不但是可能的，也是相对较为容易的。

参考资料

- 触摸和鼠标在一起
- Touch And Mouse

如何延时触发事件

jQuery的文档中提到性能优化的时候，有一条就是关于事件触发频率的问题，大部分影响性能的操作不是 `click` 之类的事件，而是 `scroll` ， `mouseover` 这样的事件。解决这些事件频繁触发的方法也很简单，就是加一个定时器，延时执行。

代码

```
function debounce(method, delay) {
    clearTimeout(method._tId); // important!
    method._tId = setTimeout(function() {
        method();
    }, delay);
}
var DAscrollPosition = $(window).scrollTop();

function getScrollDirection() {
    var scroll = $(window).scrollTop();
    if (scroll > DAscrollPosition) {
        parent.postMessage('scrolldown', 'http://www.tlc.com.adops-002.dp.discovery.com');
    }
    if ((scroll < 100) && (scroll < DAscrollPosition)) {
        parent.postMessage('scrollup', 'http://www.tlc.com.adops-002.dp.discovery.com');
    }
    DAscrollPosition = scroll;
};
$(window).scroll(function() {
    debounce(getScrollDirection, 100);
});
```

数组的操作

NodeList

NodeList看起来像一个Array（数组），你可以使用中括号来访问他们的节点，而且你还可以通过length属性知道它有多少元素。但是它并没有实现Array的所有接口，因此使用 $$('*').forEach 会返回错误，在JavaScript的世界里，有一堆看起来像Array但其实不是的对象。如function中的arguments对象。因此在他们身上通过call和apply来应用数组的方法是非常有用的。

简单的将他们转换为数组的方法, 比如函数的参数arguments:

```
function func() {
    var args = Array.prototype.slice.call(arguments);
    // or
    var args = [].slice.call(arguments);
}
```

基本操作

常规的构造方法是循环遍历赋值，取值一般直接使用索引。常用的方法有:

- push(): Adds one or more elements to the end of an array and returns the new length of the array.
- pop(): Removes the last element from an array and returns that element.
- shift(): Removes the first element from an array and returns that element.
- reverse(): Reverses the order of the elements of an array — the first becomes the last, and the last becomes the first.
- sort(): Sorts the elements of an array in place and returns the array.
- splice(): Adds and/or removes elements from an array.
- unshift(): Adds one or more elements to the front of an array and returns the new length of the array.

向数组中插入指定元素

- 使用索引值： 插入或替换元素
- push: 末尾添加元素
- unshift: 首部添加元素

特殊的，要提到一个`splice`函数，使用方法:

```
array.splice(start, deleteCount[, item1[, item2[, ...]]])
```

参数需要注意一下:

- start： 开始删除的位置
- deleteCount： 删除的个数
- [, item1[, item2[, ...]]]： 替换的元素, 如果start超过了数组长度，实际上就是追加的效果，删除的元素小于添加的元素时候，也是追加效果。

删除数组中指定元素

删除首尾的方式是使用: `pop` 和 `shift` 。

但是如果我们要删除特殊的指定元素，先要获取到指定元素的下标，然后使用 `splice` 进行替换。

ES5标准中新增的方法

- forEach (js v1.6)
- map (js v1.6)
- filter (js v1.6)
- some (js v1.6)
- every (js v1.6)
- indexOf (js v1.6)
- lastIndexOf (js v1.6)
- reduce (js v1.8)
- reduceRight (js v1.8)

对于让人失望很多次的IE6-IE8浏览器，Array原型扩展可以实现以上全部功能，所以也就有了ES5-shim项目。马上都要到ES6得节奏了，这些再不熟悉就别做前端了。

forEach

简化遍历的方法，使用起来是这样: `arr.forEach(callback[, thisArg])` 。callback默认带有三个参数:

- currentValue
- index
- array

thisArg是可选参数，代表可选的上下文参数（改变回调函数里面的this指向）.如果这第2个可选参数不指定，则使用全局对象代替（在浏览器是为window），严格模式下甚至是undefined.

注意: 和使用for循环遍历的区别是， forEach不会遍历纯粹的没定义的元素， 比如：

```
var arr = [1,2,,4];

arr.forEach(console.log);
for (var i = 0, len = arr.length; i < len; i++) {
    console.log(arr[i]);
}
```

every

按照条件判断数组是否符合要求的作用,返回的是Boolean值，它会用回调函数去遍历所有的数组元素，一般需要制定return的条件。如不指定，默认返回false。找出所有满足的情况，不满足就退出执行。返回值只要是弱等于== true/false就可以了，而非非得返回 === true/false.

some

按照条件判断数组是否符合要求的作用,返回的是Boolean值，它会用回调函数去遍历所有的数组元素，只要有符合的，就会返回true。如果回调函数不指定return值，默认返回false。效果就和forEach类似了。返回值只要是弱等于== true/false就可以了，而非非得返回 === true/false.

fliter

过滤，筛选的作用。返回的是一个新数组，根据回调函数去判断筛选元素。返回值只要是弱等于== true/false就可以了，而非得返回 === true/false.

map

映射的作用，将原数组的每个元素通过回调方法映射，返回一个同样大小的新数组。

```
var data = [1, 2, 3, 4];

var arrayOfSquares = data.map(function (item) {
  return item * item;
});

alert(arrayOfSquares); // 1, 4, 9, 16
```

reduce

用法接近于迭代，递归。语法为: `array.reduce(callback[, initialValue])` .这个方法是ES5.1之后提出的，算是比上面的方法都新一点。

callback函数接受4个参数：之前值、当前值、索引值以及数组本身。initialValue参数可选，表示初始值。若指定，则当作最初使用的previous值；如果缺省，则使用数组的第一个元素作为previous初始值，同时current往后排一位，相比有initialValue值少一次迭代。

```
var sum = [1, 2, 3, 4].reduce(function (previous, current, index, array) {
    console.log(previous, current, index, array);
    return previous + current;
});
console.log(sum); // 10
```

如果 `initialValue` 不存在，那么索引就从2开始，以后每次传递的previous就是上一个函数return的值。

如果 `initialValue` 存在，那么传入的 `initialValue` 就作为第一次的 `previous` 值。

reduceRight

`reduceRight` 跟 `reduce` 相比，语法类似： `array.reduceRight(callback[, initialValue])` 。

差别在于 `reduceRight` 是从数组的末尾开始的。

参考资料

- MDN: Array
- ES5中新增的Array方法详细说明
- ES5-shim
- Javascript – Arraylike的7种实现

对象的操作

判断元素是否存在对象中

使用 `for..in` 可以遍历元素的所有属性，如果是带有继承的对象，配合 `hasOwnProperty` 遍历出对象自身的属性，而不是原型链上的。

对象在数组中

比如这样:

```
var arr = [
    {order: 3, name: 3},
    {order: 1, name: 1},
    {order: 2, name: 2}
]
```

如何遍历

一般数组的遍历还是采取循环比较，返回index索引。

如何排序

排序的话，使用Array自带的sort方法，传入自定的比较对象，比如：

```
arr.sort(function(a,b) {
    return a.order > b.order? false: true;
})

/*
arr = [
    {order: 3, name: 3},
    {order: 2, name: 2},
    {order: 1, name: 1}
]
 */
```

参考资料

- MDN: Object

mobile设备的横竖屏切换检测

orientationchange Event

我们可以选择在window对象上监听 `orientationchange` 属性。比如:

```
// Listen for orientation changes
window.addEventListener("orientationchange", function() {
    // Announce the new orientation number
    alert(window.orientation);
    alert(window.outerWidth);
    alert(innerWidth);
}, false);
```

window.orientation的取值：

- 0: portrait
- 90: landscape rotated to the left
- -90: landscape rotated to the right
- 180: portrait, ipad下才能取到

resize Event

设备旋转的时候，window的resize事件也是被触发的。判断的方法是:

- outerWidth, outerHeight: 检测是point
- innerWidth, innerHeight: 检测的是pixel

window.matchMedia

这是一段JS代码，可以查询对应的css媒体查询语句是否匹配。

```
// Find matches
var mql = window.matchMedia("(orientation: portrait)");

// If there are matches, we're in portrait
if(mql.matches) {
    // Portrait orientation
} else {
    // Landscape orientation
}

// Add a media query change listener
mql.addListener(function(m) {
    if(m.matches) {
        // Changed to portrait
    }
    else {
        // Changed to landscape
    }
});
```

Media Query

使用css媒体查询:

```
/* portrait */
@media screen and (orientation:portrait) {
    /* portrait-specific styles */
}
/* landscape */
@media screen and (orientation:landscape) {
    /* landscape-specific styles */
}
```

参考资料

- Detect Orientation Change on Mobile Devices

生成随机数字

Math.random, returns a floating-point, pseudo-random number in the range [0, 1)

[0,1)

```
// Returns a random number between 0 (inclusive) and 1 (exclusive)
function getRandom() {
  return Math.random();
}
```

[min, max)

```
// Returns a random number between min (inclusive) and max (exclusive)
function getRandomArbitrary(min, max) {
  return Math.random() * (max - min) + min;
}
```

[min, max)整数

```
// Returns a random integer between min (included) and max (excluded)
// Using Math.round() will give you a non-uniform distribution!
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}
```

参考资料

- Math random

setTimeout的误区

很多人对 setTimeout 函数的理解就是：延时为 n 的话，函数会在 n 毫秒之后执行。事实上并非如此,这里会存在几个问题:

setTimeout 函数的及时性问题

```
var d = new Date, count = 0, f, timer;
timer = setInterval(f = function (){
    if(new Date - d > 1000)
        clearInterval(timer), console.log(count);
    count++;
}, 0);
```

可以看出 1s 中运行的次数大概在 200次 左右，有人会说那是因为 new Date 和 函数作用域的转换消耗了时间,其实不是，而是setInterval 和 setTimeout 函数运转的最短周期是 5ms 左右，这个数值在 HTML规范 中也是有提到的:

```
5. Let timeout be the second method argument, or zero if the argument was omitted.
如果 timeout 参数没有写，默认为 0
7. If nesting level is greater than 5, and timeout is less than 4, then increase timeout to 4.
如果嵌套的层次大于 5 ，并且 timeout 设置的数值小于 4 则直接取 4.
```

如果需要更加短的周期，可以使用:

- requestAnimationFrame 它允许 JavaScript 以 60+帧/s 的速度处理动画，他的运行时间间隔比 setTimeout 是要短很多的。
- process.nextTick 这个是 NodeJS 中的一个函数，利用他可以几乎达到上面看到的 while 循环的效率
- ajax 或者 插入节点 的 readState 变化
- MutationObserver
- setImmediate

会被阻塞

由于Javascript是单线程的，所以会存在被阻塞的情况:

```
var d = new Date;
setTimeout(function(){
    console.log("show me after 1s, but you konw:" + (new Date - d));
}, 1000);
while(1) if(new Date - d > 2000) break;
```

我们期望 console 在 1s 之后出结果，可事实上他却是在 2075ms 之后运行的，这就是 JavaScript 单线程给我们带来的烦恼，while循环阻塞了 setTimeout 函数的执行。

无法捕获

try...catch捕捉不到它的错误:

```
try{
    setTimeout(function(){
        throw new Error("我不希望这个错误出现！")
```

```
    }, 1000);
} catch(e){
    console.log(e.message);
}
```

在与DOM事件打交道的时候

因为浏览器端主要由三个线程：javascript执行，UI渲染，事件触发队列。javascript执行的线程与UI渲染的线程又是互斥的，所以如果在一个dom事件，特别是`onmousexxx`和`onkeyxx`事件中，对另外的dom元素进行操作，比如`focus`的时候，需要设置`setTimeout`将这些操作添加到事件触发的队列中，等当然的dom操作执行完成后执行。

比如这个代码:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>setTimeout</title>
    <script type="text/javascript">
        function get(id) {
            return document.getElementById(id);
        }
        window.onload = function () {
            //第一个例子：未使用setTimeout
            var makeBtn = get('makeinput');
            makeBtn.onmousedown = function (e) {
                console.log(e.type);
                var input = document.createElement('input');
                input.setAttribute('type', 'text');
                input.setAttribute('value', 'test1');
                get('inpwrapper').appendChild(input);
                input.onfocus = function (e) {//观察我们新生成的input什么时候获取焦点的，或者它有没有像原文作者说的那样被丢弃了
                    console.info('input focus');
                };
                input.focus();
                input.select();
            }
            makeBtn.onclick = function (e) {
                console.log(e.type);
            };
            makeBtn.onmouseup = function (e) {
                console.log(e.type);
            };
            makeBtn.onfocus = function () {//观察我们生成按钮什么时候获取焦点的
                console.log('makeBtn focus');
            }

            //第二个例子：使用setTimeout
            var makeBtn2 = get('makeinput2');
            makeBtn2.onmousedown = function (e) {
                console.log(e.type);
                var input = document.createElement('input');
                input.setAttribute('type', 'text');
                input.setAttribute('value', 'test1');
                get('inpwrapper2').appendChild(input);
                input.onfocus = function (e) {//观察我们新生成的input什么时候获取焦点的，或者它有没有像原文作者说的那样被丢弃了
                    console.info('input focus');
                };
                //setTimeout
                setTimeout(function () {
                    input.focus();
                    input.select();
                }, 0);
            }
            makeBtn2.onclick = function (e) {
                console.log(e.type);
            };
            makeBtn2.onmouseup = function (e) {
                console.log(e.type);
            };
```

```
            makeBtn2.onfocus = function () {//观察我们生成按钮什么时候获取焦点的
                console.log('makeBtn2 focus');
            }
            //第三个例子，onkeypress输入的时候少了一个值
            get('input').onkeypress = function () {
                get('preview').innerHTML = this.value;
            }
        }
    </script>
</head>
<body>
    <h1><code>setTimeout</code></h1>
    <h2>1、未使用 <code>setTimeout</code></h2>
    <button id="makeinput">生成 input</button>
    <p id="inpwrapper"></p>


    <h2>2、使用 <code>setTimeout</code></h2>
    <button id="makeinput2">生成 input</button>
    <p id="inpwrapper2"></p>


    <h2>3、另一个例子</h2>
    <p>
        <input type="text" id="input" value="" /><span id="preview"></span>
    </p>
</body>
</html>
```

你能说出点击对应的输出结果吗，并告知原因？

参考资料

- JavaScript异步编程原理
- JavaScript秘密花园
- javascript线程解释（setTimeout,setInterval你不知道的事）

前端面试

参考资料

- Front-end-Developer-Interview-Questions

前端技能图谱

一些大前端，小前端，前端技能的脑图，见assets目录。当好一个前端工程师，特别重要的一点是要理解浏览器的工作原理。这一块可以参考前面的原理章节。

- The Web platform: Browser technologies
- web开发技能树,英文: web开发技能树，和魔兽世界里加天赋类似的效果。
- Web 开发技能树: 上面那个的中文版，多了些图书资料。
- JS Recipes: JavaScript tutorials for backend and frontend development.
- Github上最火的前端开源项目: 从结构介绍到部署。
- 怎样成长为一个优秀的 Web 前端开发工程师？
- web前端学习笔记
- 结合个人经历总结的前端入门方法

培训公司

一般的培训机构都会列出基本的技能，也是值得我们学习的一部分。

- 珠峰培训
- 智能社
- 妙趣课堂
- 极客学院
- 慕课网
- 实验楼

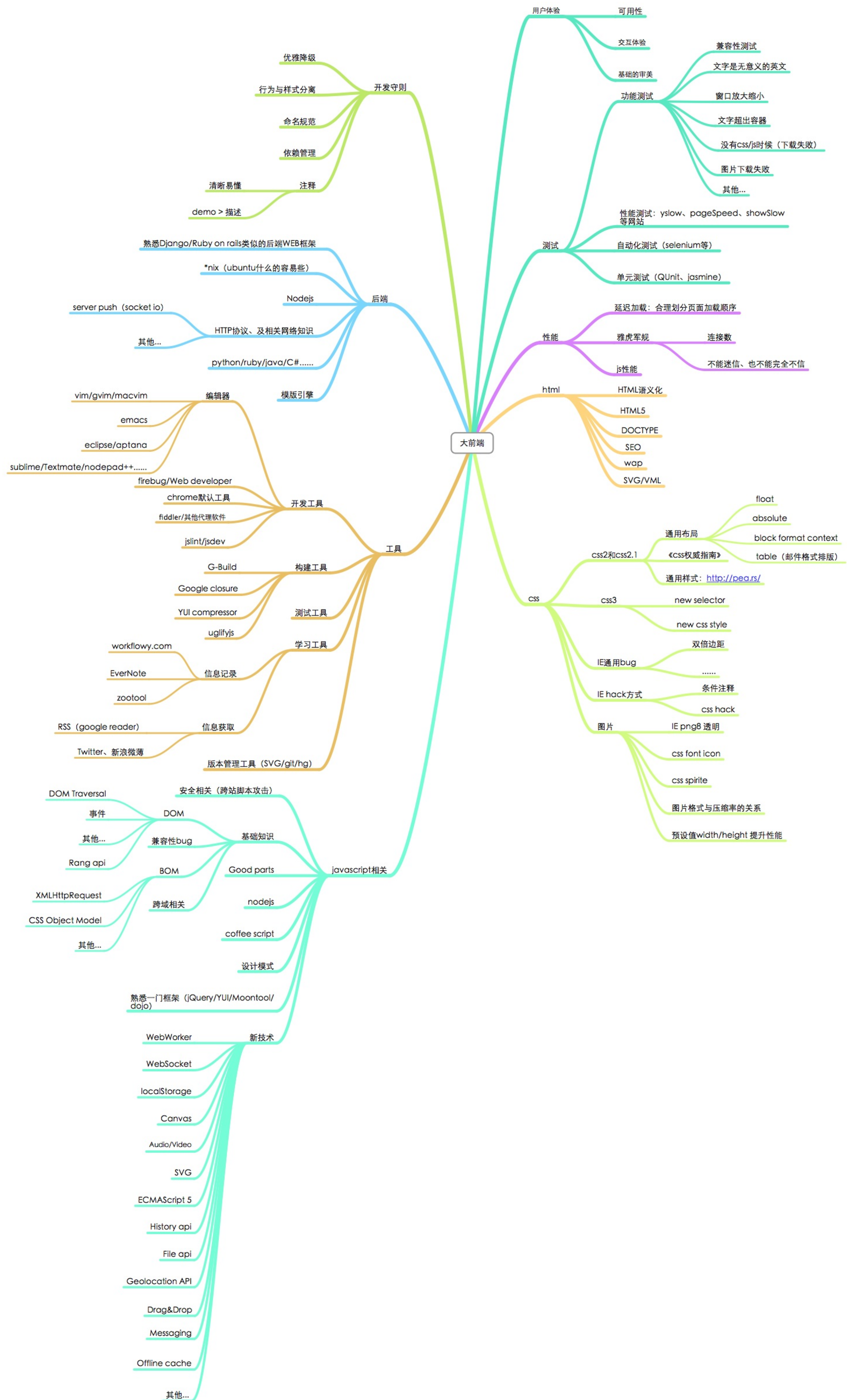
面试集合

- 5 Typical JavaScript Interview Exercises
- Front-end-Developer-Interview-Questions
- 最全前端面试问题及答案总结
- 前端面试笔试题们

web历史

- Web开发的发展史
- Web 研发模式演变

脑图


大前端
用户体验
可用性
交互体验
基础的审美
测试
功能测试
兼容性测试
文字是无意义的英文
窗口放大缩小
文字超出容器
没有css/js时候（下载失败）
图片下载失败
其他...
性能测试：yslow、pageSpeed、showSlow等网站
自动化测试（selenium等）
单元测试（QUnit、jasmine）
性能
延迟加载：合理划分页面加载顺序
雅虎军规
连接数
不能迷信、也不能完全不信
js性能
html
HTML语义化
HTML5
DOCTYPE
SEO
wap
SVG/VML
css
css2和css2.1
通用布局
float
absolute
block format context
table（邮件格式排版）
《css权威指南》
通用样式：http://pea.rs/
css3
new selector
new css style
IE通用bug
双倍边距
......
IE hack方式
条件注释
css hack
图片
IE png8 透明
css font icon
css spirite
图片格式与压缩率的关系
预设值width/height 提升性能
开发守则
优雅降级
行为与样式分离
命名规范
依赖管理
注释
清晰易懂
demo > 描述
后端
熟悉Django/Ruby on rails类似的后端WEB框架
*nix（ubuntu什么的容易些）
Nodejs
HTTP协议、及相关网络知识
server push（socket io）
其他...
python/ruby/java/C#......
模版引擎
工具
编辑器
vim/gvim/macvim
emacs
eclipse/aptana
sublime/Textmate/nodepad++......
开发工具
firebug/Web developer
chrome默认工具
fiddler/其他代理软件
jslint/jsdev
构建工具
G-Build
Google closure
YUI compressor
uglifyjs
测试工具
学习工具
信息记录
workflowy.com
EverNote
zootool
信息获取
RSS（google reader）
Twitter、新浪微薄
版本管理工具（SVG/git/hg）
javascript相关
安全相关（跨站脚本攻击）
基础知识
DOM
DOM Traversal
事件
其他...
Rang api
兼容性bug
BOM
XMLHttpRequest
CSS Object Model
其他...
跨域相关
Good parts
nodejs
coffee script
设计模式
熟悉一门框架（jQuery/YUI/Moontool/dojo）
新技术
WebWorker
WebSocket
localStorage
Canvas
Audio/Video
SVG
ECMAScript 5
History api
File api
Geolocation API
Drag&Drop
Messaging
Offline cache
其他...


前端工程师
Web服务
YQL/Company Open Platform
APP Engine/Cloud
Google Open API/Github
前沿技术社区/会议
YDN/YUIConf/JSConf
QCon/Velocity
D2/WebRebuild
NodeParty/HTML5研究小组
UED Teams
移动终端
jQueryMobile/html5/css3
iPhone/iPad/iOs/android
responsive UI Design
安全
CSRF/XSS
ADsafe/Caja/FBJS/Sandbox
软技能
知识管理/总结分享
沟通技巧/团队开发/需求管理/PM
代码模块化开发/代码版本管理
交互设计/可用性/可访问性知识
开发流程/部署
JSLint/CSSLint/YUICompressor
JSMin/TPacker-minifier
Ant/Make
JSDoc/YUIDoc
LAMP
前端框架/库
jQuery/YUI2/YUI3
Prototype/Mootools/ExtJS
Smarty/Django/Zend
YUITest/QUnit/Jasmine
前端测试/前端MVC
浏览器兼容性
IE6/7/8/9
Firefox3.6/4/5/6/7/8...
Chrome/Opera/Safari
切页面
HTML
CSS
PhotoShop
编程语言
JavaScript
PHP/Python/Perl/Ruby/BashShell...
NodeJS/FlashAS
调试工具
Firebug/Firebug-lite/Web Inspector
YSlow/Smushit
IEDeveloperToolBar/IETester
SuperPreview/JsBeautifier
Fiddler/WireShark/tcpdump
开发工具
VIM/Aptana/Notepad++/EditPlus...
SygWin/svn/git
计算机知识储备
编译原理/计算机网络
操作系统/算法原理
软件工程/软件测试原理
前端标准/规范
w3c/DOM/BOM/XHTML/XML/JSON/JSONP
HTML5/CSS3
ECMAScript3/ECMAScript5
CommonJS/AMD
HTTP1.1
性能
WebPageTest
ShowSlow/YSlow/34Rule
PageSpeed
HttpWatch
DynaTrace's Ajax
编程知识储备
数据结构/MVC
面向对象/切面编程
函数式编程
JavaScript设计模式
JavaScript编程模式
@拔赤
weibo.com/jayli

前端知识点

HTML+CSS

```
对Web标准的理解、浏览器内核差异、兼容性、hack、CSS基本功：布局、盒子模型、选择器优先级及使用、HTML5、CSS3、移动端
```

- 一些自适应或垂直水平居中问题汇总
- Normal Flow
- Containing Block
- Margin Collapse
- BFC
- Baseline
- Writing Mode
- unicode-bidi

JavaScript

补充一下JS的深入理解，参考汤姆大叔的博客: 深入理解JavaScript系列，理解一些原理性质的东西。别停留在使用上，不然面试的时候很容易吃亏。那些你觉得不在意的东西，往往会是你成功的阻碍。

```
数据类型、面向对象、继承、闭包、插件、作用域、跨域、原型链、模块化、自定义事件、内存泄漏、事件机制、异步装载回调、模板引擎、Nodejs、JSON、aj
```

JS的知识点，我很喜欢王子墨总结的那些，很完善的感觉：

```
1、DOM结构 —— 两个节点之间可能存在哪些关系以及如何在节点之间任意移动。

2、DOM操作 —— 如何添加、移除、移动、复制、创建和查找节点等。

3、事件 —— 如何使用事件，以及IE和标准DOM事件模型之间存在的差别。

4、XMLHttpRequest —— 这是什么、怎样完整地执行一次GET请求、怎样检测错误。

5、严格模式与混杂模式 —— 如何触发这两种模式，区分它们有何意义。

6、盒模型 —— 外边距、内边距和边框之间的关系，及IE8以下版本的浏览器中的盒模型

7、块级元素与行内元素 —— 怎么用CSS控制它们、以及如何合理的使用它们

8、浮动元素 —— 怎么使用它们、它们有什么问题以及怎么解决这些问题。

9、HTML与XHTML —— 二者有什么区别，你觉得应该使用哪一个并说出理由。

10、JSON —— 作用、用途、设计结构。
```

基础知识

- JavaScript的基本数据类型
- 内置对象的常用方法
- 理解事件机制
- 理解原型继承
- 理解作用域问题

- 理解模块化
- 性能优化
- 知道基本的编程语法，比如循环，判断，try/catch 等等
- 理解包括多种函数定义以及赋值的方式，包括匿名函数
- 理解基本的命名空间，全局（window）空间以及对象空间（不包括闭包）
- 理解上下文的角色以及 this 变量的使用
- 理解各种对象以及函数的初始化和声明方式
- 理解 javascript 比较操作符，如<, >, ==, ===，以及对象和字符串比较的原理和对象映射
- 理解对象属性和函数的数组索引，以及这和真实的数组之间的区别。

中级知识

- 理解常用库的实现原理，比如选择器部分，事件绑定部分
- 检测浏览器类型与版本
- 了解特性检测
- 理解定时器，以及它的工作原理，包括何时以及如何使用定时器来异步执行方法调用
- 关于回调的深度支持，以及如何通过 call 和 apply 方法来控制上下文和函数参数传递
- 理解 JSON 标记以及 eval 函数
- 理解闭包以及他们如何影响你的代码效率
- AJAX 以及对象序列化

高级知识

- 理解方法的 arguments 变量，包括如何使用它来通过 arguments.length 重载函数，以及通过 arguments.callee 来进行递归调用，需要注意使用这个特性有一定的危险性，因为 ECMAScript 5 的 Strict 模式不支持此功能，但 jQuery 和 Dojo 都用到了它。
- 高级闭包比如 self-memoizing 函数，partially applied 函数，以及最可爱的 (function(){})() 调用。
- 函数以及 HTML prototype，prototype chain，以及如何使用基本的javascript对象和函数（比如 Array）来简化代码。
- 对象类型以及 instanceof 的使用
- 正则表达式和表达式编译
- With 语句以及为什么不要使用它们
- 最困难的部分，知道如果利用所有这些工具，并产生处干净，整洁，健壮，快速，可维护以及兼容不同浏览器的代码。

和**web**相关的知识

- 如何高效的操作 Dom（添加，删除以及更新），还有如何通过使用 document fragments 这样的工具来最小化浏览器的 re-flows。
- 夸浏览器的 DOM 元素属性提取（比如，style，position等等），jQuery 和 Dojo 都可以很好的完成这些工作，尽管如此，理解从 CSS 和 style 标签中提取属性的差异，以及如何计算 position 和 size 还是很重要的。
- 夸浏览器的事件处理，绑定，反绑定，冒泡，以及如何取得期望的回调上下文。在一次，现成的框架也可以很好的处理这些事情，但是你应该对 IE 浏览器和 W3C 标准浏览器之间的不同有所了解。
- 正则表达式选取 DOM 节点
- 浏览器功能检测以及智能降级

其他

```
HTTP、WEB安全、正则、优化、重构、响应式、团队协作、可维护、SEO、UED、架构、职业生涯
```

参考资料

- 我所了解的CSS
- 你不知道你不懂 javascript
- 代码之谜
- 汤姆大叔的博客: 深入理解JavaScript系列
- JavaScript核心

原则与技巧

面试方式

一般程序员的面试分为：代码笔试+问答面试。可能有一面，二面，三面以及多面的情况。迁移两轮主要还是技术方面的为主，由浅入深。三四轮的话主要就是hr和boss和你聊待遇和规划了。

代码笔试主要就是考察基本能力，问答主要是对项目以及个人技能的深入了解。

面试人员应该具备的技能

一个优秀的程序员应该具有怎样的技能:

- 基础扎实
- 主动思考
- 爱学习
- 有深度
- 有视野

往细致了点说，就是不要停留在使用的层面，多余了解更深层的原理。

关于题目

什么样的面试题是好的？淘宝大神wintercn认为有三点衡量指标:

- 区分度
- 深度
- 覆盖范围

是的，请注意这里并没有使用“难度”这个词，因为这三个指标都与难度有关系。这个题的答案可以分成不同的层级:

- position属性常用的取值static、relative以及absolute和它们的基本行为是每个前端都应该掌握的。这包括relative和absolute的定位原点。 -fixed旧版本IE不支持，但是一个对技术有热情的工程师也是应该了解的。 -有过研究工程师可以知道absolute的containing block计算方式跟正常流不同，当然如果没读过标准的话，表述方式不一定是这样。 -对CSS布局有深入研究的工程师会知道position跟display、margin collapse、overflow、float这些特性相互叠加后的行为。

区分度可以让题目可以适用于入门级到专家级的各种面试者，深度可以保证有深度研究的面试者可以展示他们的才能，覆盖范围可以有效地了解面试者擅长的方向。

当面试者前面回答的答案足够完美，我就会进行追问，确保问到我开始不懂或者面试者开始不懂为止，这样可以大大延展题目的区分度和深度。

考察能力

面试应该更注重“考察能力”。这个能力应该是:

```
web前端工程师的竞争力 = web前端知识 + 能力
能力 = 编程能力 + 工程能力 + 架构能力
```

这其中不包括所谓的学习能力，因为我认为学习能力是通过已有知识来体现的，如果一个具有超强学习能力的人来应聘web前端工程师但是他具有如此强的学习能力却连position这么重要的属性都没学会，那是不是下一步该要求这人附上证明自己没有精神疾病的诊断书？

工程能力和架构能力一般针对层级较高的工程师，所以一般来讲所谓能力考察就是编程能力，然后呢，编程能力一般考查方式就是案例问题，也就是传说中的——“算!法!题!”（当然我曾提到，它们与其说是算法题，不如说是稍微复杂点的小程序，它们之所以看上去不太有用是因为出题的人为了避免理解麻烦剥掉了实际的业务场景，毕竟各个公司的业务都不是一句两句可以讲清楚的）。

关于评判

面试中未必是所有题目全都回答"正确"就一定会通过或者较高评价。面试是面试官和面试者双方"挖掘与展示才能"的过程，参考前面提到的面试过程，全部回答正确的情况很可能是因为面试官不感兴趣懒得追问。

对于面试官而言，基本评判原则就是"我要不要这个人做我的同事？"，多数情况下，这个答案会非常清楚。一些题目是充分的，也就是"回答对了说明这个人具有可以依靠的才能"，一些题目则是必要的，也就是"回答错了说明这个人无法胜任我们的工作"。

在position一题的评判上，我一般认为能够答对static、relative以及absolute就已经可以达到必要标准。而因为CSS layout可能是面试官最擅长的部分，又考虑到误差，当面试者能回答80%以上的追问，基本就能判定面试者水平远高于主考官，在工作中能够作为CSS方面的专家来依靠。

一些人说"属性可以google搜索"则更离谱，position在CSS布局中是相当基础的知识，对它的行为理解深度实际上代表了一个工程师对于CSS布局系统的理解，这个理解需要长时期的学习，绝对不是可以临时google得来的。正如同考人英语，若是不认识visibility尚可以解释说确实没怎么用过，而不会写英文字母v则说明这个人根本没学过英语。

STAR面试法

这里提到了一种面试原则，叫做STAR面试法.STAR”是SITUATION（背景）、TASK（任务）、ACTION（行动）和RESULT（结果）四个英文单词的首字母组合。

在招聘面试中，仅仅通过应聘者的简历无法全面了解应聘者的知识、经验、技能的掌握程度及其工作风格、性格特点等方面的情况。而使用STAR技巧则可以对应聘者做出全面而客观的评价。

- 背景（SITUATION）: 通过不断提问与工作业绩有关的背景问题，可以全面了解该应聘者取得优秀业绩的前提，从而获知所取得的业绩有多少是与应聘者个人有关，多少是和市场的状况、行业的特点有关。
- 工作任务（TASK）: 每项任务的具体内容是什么样的。通过这些可以了解应聘者的工作经历和经验，以确定他所从事的工作与获得的经验是否适合所空缺的职位。
- 行动（ACTION）: 即了解他是如何完成工作的，都采取了哪些行动，所采取的行动是如何帮助他完成工作的。通过这些，可以进一步了解他的工作方式、思维方式和行为方式。
- 结果（RESULT）: 每项任务在采取了行动之后的结果是什么，是好还是不好，好是因为什么，不好又是因为什么。

面试技巧

基础知识

对基础部分的内容掌握必须牢靠，什么属性什么方法的，都要知道到底是什么。

项目经验

面试过程其实不是一个你问我答的情况，正规一点的面试还是要以你的实际接触为点，扩展开来对你考核。所以在讲项目的时候，你需要展示你自己的亮点，可以说一些装逼的词，但装逼也是得有真材实料的。比如我在项目中使用了WebSocket，

那么面试官很可能问你WebSocket是什么，底层原理你知道么？如果你当场傻掉，面试官就会觉得你只是会使用别人的东西，并不在意实现原理，终究是码农。那么事先你就应当去看看WebSocket协议的官方文档(纯英文，看得累死我了！)，这样面试官一问你，你能头头是道，会大大加分。再比如，你在项目中使用了模块化，那么你就一定要知道什么是模块化，而不是说你会用模块化工具。其实要求并不高，你只要能很好说清楚什么是AMD规范，什么是CommonJs规范，各自的优缺点是什么就很够了。

记住重要的一点是，一定要把面试官往你熟悉的领域引导，这真的很重要，因为如果你不引导，面试官不了解你的项目，看不到你的亮点，就只能一直问技术问题刁难你，人家在大公司待这么久了，还不是轻松碾压你。所以你在引导的同时，时不时提及一些事先准备好的关键词，技术官一问，你一回答，怎么都妥了。

当然，如果有些问题是你可能是真的不会的，但是也不要出现好像是、可能是、我猜之类的词眼(我之前就是这样跪掉的)，而是说按照我的理解、给过一点思考时间、我不太懂这个问题需要我从哪个角度解析、我以前遇到类似的问题是怎样这个问题应该也是这样...这样给面试官的印象是，即便你不懂，但是你在全力思考，而且这样会给自己争取很多时间。

hr面谈

有的公司其实技术主管也就决定了你的待遇问题，所以这一环节不一定出现。但是问的几个问题大致可以提前思考一下:

- 你为什么离职
- 你为什么选择我们
- 用几个词描述一下自己
- 期望的待遇是如何定的
- 你是如何规划自己的

对公司提问

这一块也是最后的环节，一般会让你问几个问题，我自己会问的有:

- 公司的技术团队规模和方向
- 员工的晋升途径
- 公司的作息与加班情况，以及补偿情况
- 公积金是按什么比例缴纳

总结

- 面试题目： 根据你的等级和职位变化，入门级到专家级：广度↑、深度↑。
- 题目类型： 技术视野、项目细节、理论知识，算法，开放性题，工作案例。
- 细节追问： 可以确保问到你开始不懂或面试官开始不懂为止，这样可以大大延展题目的区分度和深度，知道你的实际能力。因为这种关联知识是长时- 期的学习，绝对不是临时记得住的。
- 态度：回答问题再棒，面试官（可能是你面试职位的直接领导），会考虑我要不要这个人做我的同事？所以态度很重要。（感觉更像是相亲）
- 机会总是留给有准备的人，每一次都要好好对待
- 别紧张，说话的时候条理清晰

参考资料

- 如何面试前端工程师？
- 一名靠谱的JavaScript程序员应备的素质
- FEX面试原则
- STAR面试法

html/css面试题

HTML

- 每个HTML文件里开头都有个很重要的东西，Doctype，知道这是干什么的吗？
- div+css的布局较table布局有什么优点？
- strong与em的异同？
- 你能描述一下渐进增强和优雅降级之间的不同吗?
- 为什么利用多个域名来存储网站资源会更有效？
- 请描述一下cookies，sessionStorage和localStorage的区别？
- 简述一下src与href的区别。
- 你如何理解HTML结构的语义化？

CSS

- 有哪项方式可以对一个DOM设置它的CSS样式？
- CSS都有哪些选择器？
- CSS选择器的优先级是怎么样定义的？
- CSS中可以通过哪些属性定义，使得一个DOM元素不显示在浏览器可视范围内？
- 超链接访问过后hover样式就不出现的问题是什么？如何解决？
- 行内元素和块级元素的具体区别是什么？行内元素的padding和margin可设置吗？
- css中可以让文字在垂直和水平方向上重叠的两个属性是什么？
- px和em的区别。
- 描述一个”reset”的CSS文件并如何使用它。知道normalize.css吗？你了解他们的不同之处？

参考资料

- BAT及各大互联网公司2014前端笔试面试题：HTML/CSS篇

JavaScript面试题

JavaScript面试的题目，主要考察的内容还是逻辑性为主。基础的部分，主要是对内置函数的使用，比如String, Math, Array对象。还有一个比较讨面试喜欢的，就是正则表达式。

作用域链

JS核心部分需要理解的一个重要部分。

闭包

原始对象

```
var a = 1;
a.a = 2;
console.log(a.a);
```

这一题需要理解的是点的作用，在JS解释器中，首先会判断左侧的变量是什么类型，如果是普通对象，会创建一个新的对象作用域，然后挂载a属性。console.log部分的a也是又创建了一个封装对象，但是这个对象下面的a是没有赋值的。

引用类型

```
var a = {n:1};
var b = a;
a.x = a = {n:2};

alert(a.x);// --> undefined
alert(b.x);// --> [object Object]
```

这个涉及到了连续赋值的情况，详情参考[javascript 连等赋值问题](javascript 连等赋值问题).在程序运行到之后，先确定好了 a.x 和 a 的引用，再从右往左开始赋值的。还有一点就是可以理解的是 `.` 运算优先级高于 `=` 运算符，所以会先创建 `a.x` 对象，然后在执行赋值过程。赋值顺序从右向左。

原型链

JS核心部分的另一个需要理解的重要部分。

类型转换

比较时候的转换的原则:

- 一个是number一个是string时，会尝试将string转换为number
- 尝试将boolean转换为number，0或1
- 尝试将Object转换成number或string，取决于另外一个对比量的类型

运算过程的转换原则：

- 字符串与数字相加，变成字符串
- 字符串与数字相减，变成数字

输出下面几个代码

```
function foo1(a){
    return a + '01';
}

foo1(01);
```

```
function foo2(a){
    return a + '010';
}

foo2(010);
```

```
console.log(0.2 + 0.4);
```

```
var foo = "11"+2-"1";
console.log(foo);
console.log(typeof foo);
```

基础的代码使用技巧

- 生成[x,y]范围的随机整数
- 已知数组var stringArray = [“This”, “is”, “Baidu”, “Campus”]，Alert出”This is Baidu Campus”
- 已知有字符串foo=”get-element-by-id”,写一个function将其转化成驼峰表示法”getElementById”
- var numberArray = [3,6,2,4,1,5]; 实现倒排，排序。
- 怎样添加、移除、移动、复制、创建和查找节点
- 将一个 `#fffff` 类型的数据转换为 `rgb(255,255,255)` 形式

正则表达式

去除字符串中的多余空格？

如果使用了 `\s` 作为匹配的情况，有没有消除不了的情况（是有的），具体是什么？

为了保证页面输出安全，我们经常需要对一些特殊的字符进行转义，请写一个函数**escapeHtml**，将**<, >, &, “**进行转义

```
function escapeHtml(str) {
return str.replace(/[<>”&]/g, function(match) {
    switch (match) {
                   case “<”:
                      return “&lt;”;
                   case “>”:
                      return “&gt;”;
                   case “&”:
                      return “&amp;”;
                   case “\””:
                      return “&quot;”;
      }
  });
}
```

写一个function，清除字符串前后的空格。(兼容所有浏览器)

```
if (!String.prototype.trim) {
 String.prototype.trim = function() {
 return this.replace(/^\s+/, "").replace(/\s+$/,"");
 }
}

// test the function
var str = " \t\n test string ".trim();
alert(str == "test string"); // alerts "true"
```

中级难度的

实现一个函数clone，可以对JavaScript中的5种主要的数据类型（包括Number、String、Object、Array、Boolean）进行值复制

- 考察点1：对于基本数据类型和引用数据类型在内存中存放的是值还是指针这一区别是否清楚
- 考察点2：是否知道如何判断一个变量是什么类型的
- 考察点3：递归算法的设计

```
// 方法一：
Object.prototype.clone = function(){
        var o = this.constructor === Array ? [] : {};
        for(var e in this){
                o[e] = typeof this[e] === "object" ? this[e].clone() : this[e];
        }
        return o;
}

//方法二：
  /**
     * 克隆一个对象
     * @param Obj
     * @returns
     */
    function clone(Obj) {
        var buf;
        if (Obj instanceof Array) {
            buf = [];                    //创建一个空的数组
            var i = Obj.length;
            while (i--) {
                buf[i] = clone(Obj[i]);
            }
            return buf;
        }else if (Obj instanceof Object){
            buf = {};                   //创建一个空对象
            for (var k in Obj) {           //为这个对象添加新的属性
                buf[k] = clone(Obj[k]);
            }
            return buf;
        }else{                         //普通变量直接赋值
            return Obj;
        }
    }
```

如何消除一个数组里面重复的元素？

```
myArray.filter(function(elem, pos,self){return self.indexOf(elem)== pos;})
```

filter是过滤的意思，filter通过一个函数的参数来选择什么项需要被filter掉，函数返回true保留，false干掉。

函数参数带三个参数，第一个elem是这一项元素，第二个pos是这一项所在的位置，第三个self指的是执行filter的数组。那么，你看，巧妙吗:self.indexOf(elem) 是指这个项目在数组中的位置，位置是第一个，也就是说同样的项目在第一位和第5位都出现了，他返回的是0，而此时pos还是4，所以通过self.indexOf(elem) == pos 能判断出这一项是不是重复出现的项，如果是（返回false），则干掉它。

编写一个JavaScript函数，输入指定类型的选择器(仅需支持id，class，tagName三种简单CSS选择器，无需兼容组合选择器)可以返回匹配的DOM节点，需考虑浏览器兼容性和性能。

```
var query = function(selector) {
                var reg = /^(#)?(\.)?(\w+)$/img;
                var regResult = reg.exec(selector);
                var result = [];
                //如果是id选择器
                if(regResult[1]) {
                    if(regResult[3]) {
                        if(typeof document.querySelector === "function") {
                            result.push(document.querySelector(regResult[3]));
                        }
                        else {
                            result.push(document.getElementById(regResult[3]));
                        }
                    }
                }
                //如果是class选择器
                else if(regResult[2]) {
                    if(regResult[3]) {
                        if(typeof document.getElementsByClassName === 'function') {
                            var doms = document.getElementsByClassName(regResult[3]);
                            if(doms) {
                                result = converToArray(doms);
                            }
                        }
                        //如果不支持getElementsByClassName函数
                        else {
                            var allDoms = document.getElementsByTagName("*") ;
                            for(var i = 0, len = allDoms.length; i < len; i++) {
                                if(allDoms[i].className.search(new RegExp(regResult[2])) > -1) {
                                    result.push(allDoms[i]);
                                }
                            }
                        }
                    }
                }
                //如果是标签选择器
                else if(regResult[3]) {
                    var doms = document.getElementsByTagName(regResult[3].toLowerCase());
                    if(doms) {
                        result = converToArray(doms);
                    }
                }
                return result;
            }

            function converToArray(nodes){
                  var array = null;
                  try{
                        array = Array.prototype.slice.call(nodes,0);//针对非IE浏览器
                  }catch(ex){
                      array = new Array();
                      for( var i = 0 ,len = nodes.length; i < len ; i++ ) {
                          array.push(nodes[i])
                      }
                  }
                  return array;
          }
```

理解下sort排序的原理

数组的sort方法，默认是按照ascii排序的，为了对数字进行区分，还是手动传入一个sort函数。

```
var arr = [11,2,28,5,8,4]
arr.sort(function(a,b){return a-b})
```

sort的参数是一个排序函数，我们可以把参数a当作数组里靠后的元素，b当作数组里靠前的元素，排序函数return的值如果是正的，才执行排序，所以最后排下来是从小到大，相反，如果return的是b-a，那么就是从大到小排序。

apply和call方法的异同

对于apply和call两者在作用上是相同的，即是调用一个对象的一个方法，以另一个对象替换当前对象。将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。

但两者在参数上有区别的。对于第一个参数意义都一样，但对第二个参数： apply传入的是一个参数数组，也就是将多个参数组合成为一个数组传入，而call则作为call的参数传入（从第二个参数开始）。 如 func.call(func1,var1,var2,var3)对应的apply写法为：func.apply(func1,[var1,var2,var3]) 。

在Javascript中什么是伪数组？如何将伪数组转化为标准数组？

伪数组（类数组）：无法直接调用数组方法或期望length属性有什么特殊的行为，但仍可以对真正数组遍历方法来遍历它们。典型的是函数的argument参数，还有像调用getElementsByTagName,document.childNodes之类的,它们都返回NodeList对象都属于伪数组。可以使用Array.prototype.slice.call(fakeArray)将数组转化为真正的Array对象。

想实现一个对页面某个节点的拖曳？如何做？（使用原生JS）

- 给需要拖拽的节点绑定mousedown, mousemove, mouseup事件
- mousedown事件触发后，开始拖拽™
- mousemove时，需要通过event.clientX和clientY获取拖拽位置，并实时更新位置
- mouseup时，拖拽结束
- 需要注意浏览器边界的情况

偏门的

输出一下代码

```
function a(x){
    return function b(y){
        return y+x++
    }
}

var a1 = a(10)
var a2 = a(20)

a1(10)
a2(10)
```

```
var a = {n:1};
var b = a; // 持有a，以回查
a.x = a = {n:2};
alert(a.x);// --> undefined
alert(b.x);// --> [object Object]
```

不同的颜色标记出来页面中各层的HTML

```
[].forEach.call($ $(""),function(a){
 a.style.outline="1px solid #"+(~~(Math.random()*(1<<24))).toString(16)
 })
```

原理参考: 通过一行代码学习javascript

为什么 `++[[]][+[]]+[+[]] = 10`？

答案参考: [为什么 ++[[]][+[]]+[+[]] = 10？

参考资料

- BAT及各大互联网公司2014前端笔试面试题：JavaScript篇

jQuery面试题

如果你正要去面试一个职位，它需要你拥有多项技能，比如：Java、jQuery，它并不是希望你明白jQuery每一个细微的细节，或对其有全面的了解，但是如果你是要面试一个真正的客户端开发职位，你就需要积累更多高级的有技巧性的jQuery问题。

题目

1. jQuery 库中的 $() 是什么？
2. 网页上有 5 个
   元素，如何使用 jQuery来选择它们？
3. jQuery 里的 ID 选择器和 class 选择器有何不同？
4. 如何在点击一个按钮时使用 jQuery 隐藏一个图片？
5. $(document).ready() 是个什么函数？为什么要用它？
6. JavaScript window.onload 事件和 jQuery ready 函数有何不同？
7. 如何找到所有 HTML select 标签的选中项？
8. jQuery 里的 each() 是什么函数？你是如何使用它的？
9. 你是如何将一个 HTML 元素添加到 DOM 树中的？
10. 你能用 jQuery 代码选择所有在段落内部的超链接吗？
11. $(this) 和 this 关键字在 jQuery 中有何不同？
12. 你如何使用jQuery来提取一个HTML 标记的属性 例如. 链接的href?
13. 你如何使用jQuery设置一个属性值?
14. jQuery中 detach() 和 remove() 方法的区别是什么?
15. 你如何利用jQuery来向一个元素中添加和移除CSS类?
16. 使用 CDN 加载 jQuery 库的主要优势是什么？
17. jQuery.get() 和 jQuery.ajax() 方法之间的区别是什么?
18. jQuery 中的方法链是什么？使用方法链有什么好处？
19. 你要是在一个 jQuery 事件处理程序里返回了 false 会怎样？
20. 哪种方式更高效：document.getElementbyId("myId") 还是 $("#myId")？

参考资料

- 最常见的 20 个 jQuery 面试问题及答案

网络相关面试题

HTTP

- 解释下XMLHttpRequest
- Http的状态码
- Cache-control

参考资料

- 浅谈http中的get和post的区别
- 计算机网络协议赏析-HTTP
- 浏览器缓存机制

面试题集合

基础知识

这里有一个较为完整的面试流程可以参考:

第一部分：**Object Prototypes (**对象原型**)**

刚开始很简单。我会让侯选人去定义一个方法，传入一个string类型的参数，然后将string的每个字符间加个空格返回，例如：

```
spacify('hello world') // => 'h e l l o  w o r l d'
```

尽管这个问题似乎非常简单，其实这是一个很好的开始，尤其是对于那些未经过电话面试的侯选人——他们很多人声称精通JavaScript，但通常连一个简单的方法都不会写。

下面是正确答案，有时侯选人可能会用一个循环，这也是一种可接受的答案：

```
function spacify(str) {
      return str.split('').join(' ');
    }
```

接下来，我会问侯选人，如何把这个方法放入String对象上面，例如：

```
'hello world'.spacify();
```

问这个问题可以让我考察侯选人是否对function prototypes(方法原型)有一个基本的理解。这个问题会经常引起一些有意思的讨论：直接在对象的原型（prototypes）上添加方法是否安全，尤其是在Object对象上。最后的答案可能会像这样：

```
String.prototype.spacify = function(){
      return this.split('').join(' ');
    };
```

到这儿，我通常会让侯选人解释一下函数声明和函数表达式的区别。

第二部分：参数 **arguments**

下一步我会问一些简单的问题去考察侯选人是否理解参数（arguments）对象。我会让他们定义一个未定义的log方法作为开始：

```
log('hello world')
```

我会让侯选人去定义log，然后它可以代理console.log的方法。正确的答案是下面几行代码，其实更好的侯选人会直接使用apply.

```
function log(msg)  {
```

```
        console.log(msg);
      }
```

他们一旦写好了，我就会说我要改变我调用log的方式，传入多个参数。我会强调我传入参数的个数是不定的，可不止两个。这里我举了一个传两个参数的例子。

```
log('hello', 'world');
```

希望你的侯选人可以直接使用apply。有时人他们可能会把apply和call搞混了，不过你可以提醒他们让他们微调一下。传入console的上下文也非常重要。

```
function log(){
      console.log.apply(console, arguments);
    };
```

接下来我会让侯选人给每一个log消息添加一个"(app)"的前辍，比如：

```
'(app) hello world'
```

现在可能有点麻烦了。好的侯选人知道arugments是一个伪数组，然后会将他转化成为标准数组。通常方法是使用Array.prototype.slice，像这样：

```
function log(){
      var args = Array.prototype.slice.call(arguments);
      args.unshift('(app)');

      console.log.apply(console, args);
    };
```

第三部分：上下文

下一组问题是考察侯选人对上下文和this的理解。我先定义了下面一个例子。注意count属性不是只读取当前下下文的。

```
var User = {
  count: 1,

  getCount: function() {
    return this.count;
  }
};
```

我又写了下面几行，然后问侯选人log输出的会是什么。

```
console.log(User.getCount());

    var func = User.getCount;
    console.log(func());
```

这种情况下，正确的答案是1和undefined。你会很吃惊，因为有很多人被这种最基础的上下文问题绊倒。func是在winodw的上下文中被执行的，所以会访问不到count属性。我向侯选人解释了这点，然后问他们怎么样保证User总是能访问到func的上下文，即返回正即的值：1

正确的答案是使用Function.prototype.bind，例如：

```
var func = User.getCount.bind(User);
  console.log(func());
```

接下来我通常会说这个方法对老版本的浏览器不起作用，然后让侯选人去解决这个问题。很多弱一些的侯选人在这个问题上犯难了，但是对于你来说雇佣一个理解apply和call的侯选人非常重要。

```
Function.prototype.bind = Function.prototype.bind || function(context){
      var self = this;

      return function(){
        return self.apply(context, arguments);
      };
    }
```

第四部分：弹出窗口（**Overlay library**）

面试的最后一部分，我会让侯选人做一些实践，通过做一个'弹出窗口'的库。我发现这个非常有用，它可以全面地展示一名前端工程师的技能：HTML,CSS和JavaScript。如果侯选人通过了前面的面试，我会马上让他们回答这个问题。

实施方案是由侯选人自己决定的，但是我也希望他们能通过以下几点来实现：

在遮罩中最好使用position中的fixed代替absolute属性，这样即使在滚动的时侯，也能始终让遮罩始盖住整个窗口。当侯选人忽略时我会提示他们这一点，并让他们解释fixed和absolute定位的区别。

```
.overlay {
      position: fixed;
      left: 0;
      right: 0;
      bottom: 0;
      top: 0;
      background: rgba(0,0,0,.8);
    }
```

他们如何让里面的内容居中也是需要考察的一点。一些侯选人会选择CSS和绝对定位，如果内容有固定的宽、高这是可行的。否则就要使用JavaScript.

```
.overlay article {
      position: absolute;
      left: 50%;
      top: 50%;
      margin: -200px 0 0 -200px;
      width: 400px;
      height: 400px;
    }
```

我也会让侯选人确保当遮罩被点击时要自动关闭，这会很好地考查事件冒泡机制的机会。通常侯选人会在overlay上面直接绑定一个点击关闭的方法。

```
$('.overlay').click(closeOverlay);
```

这是个方法，不过直到你认识到点击窗口里面的东西也会关闭overlay的时侯——这明显是个BUG。解决方法是检查事件的触发对象和绑定对象是否一致，从而确定事件不是从子元素里面冒上来的，就像这样：

```
$('.overlay').click(function(e){
      if (e.target == e.currentTarget)
        closeOverlay();
    });
```

其他方面

当然这些问题只能覆盖前端一点点的知识的，还有很多其他的方面你有可能会问到，像性能，HTML5 API, AMD和CommonJS模块模型，构造函数（constructors），类型和盒子模型（box model）。根据侯选人的情况，我经常会随机提些问题。

参考资料

- 5 Typical JavaScript Interview Exercises
- Front-end-Developer-Interview-Questions
- Front-end-Developer-Interview-Questions中文版
- 最全前端面试问题及答案总结
- 前端面试笔试题们
- 前端笔试面试题
- 前端开发面试题大收集
- 收集的前端面试题和答案

参考资料

主要包含一些站点和资源。

图书资料

技能图谱

- The Web platform: Browser technologies
- web开发技能树,英文: web开发技能树，和魔兽世界里加天赋类似的效果。
- Web 开发技能树: 上面那个的中文版，多了些图书资料。
- JS Recipes: JavaScript tutorials for backend and frontend development.

HTML

- 学习CSS布局
- Dive Into HTML5
- HTML5 / CSS3 Tutorials
- 常用的HTML5、CSS3新特性能力检测写法

CSS

- scalable-css-reading-list:Collected dispatches from The Quest for Scalable CSS.
- css sharp

JavaScript

- 如何正确学习JavaScript？
- JavaScript秘密花园
- You-Dont-Know-JS: A book series on JavaScript. @YDKJS on twitter.
- Eloquent JavaScript _ Annotated ECMAScript 5.1
- The Web platform: Browser technologies: 前端工程师应该掌握的知识点。
- JavaScript高手的资源装备
- ppk-on-javascript: 国外博客大神出的书。
- Secrets of the JavaScript Ninja: 需要自己去搜pdf版本。
- JavaScript Enlightenment (PDF)
- Learning Advanced JavaScript: by John Resig, the author of jQuery. If you master this one, you’re almost done with the language.
- JavaScript The Right Way:An easy-to-read, quick reference for JS best practices, accepted coding standards, and links around the Web
- 你不知道你不懂JS

移动端JavaScript学习资料

- 司徒正美的移动web学习资料
- 响应式Web初级入门

Javascript设计模式

- JavaScript Patterns

- Learning JavaScript Design Patterns

性能优化(Web Performance Optimization)

- awesome-wpo
- superhero: 各种主题文章合集。

SVG

- awesome-svg: A curated list of SVG.

部署问题

- A field guide to Static apps: 讲解关于静态文件部署。

视频教程

我觉得看书是一种主动的过程，视频相对而言就是被动的效果，但是我觉得更加直接，我喜欢。

JavaScript

- gotoandlearn: 里面有一篇关于HTML5 Animation with TweenMax的讲解。
- imooc关于JavaScript的教程
- 极客学院上关于JavaScript的教程
- 网易云课堂上关于JavaScript的教程

设计相关

- dribbble
- codrops
- smashingmagazine
- codepen
- InVision: 设计作品标注，分享站点。
- beautifulpixels
- The Expressive Web

设计教程

- HackDesign

代码相关

- Font Awesome
- COLORS: A nicer color palette for the web.

背景图片

- geo pattern: svg
- Trianglify

效果参考

- Button effect
- link effect
- Direction-Aware Hover Effect

JavaScript库

基本库

- jQuery
- zepto
- Vanilla JS

模块加载管理

- requireJS
- Melchior.js
- webpack: 打包为静态文件，同步加载。

UI库

- polymer
- react
- riot

图片加载

- lazysizes: High performance lazy loader for images .

模板引擎

- nunjucks: mozilla出品。

jQuery插件

- jQuery Plugins: 查询jQuery插件。

MVC/MMVC

- vue.js
- Qmik: Qmik是一个面向数据接口编程,代码模块化,mvc式的开发框架,数据双向趋动,入门简单,jquery语法,快速和精简,功能强大的无线端JavaScript库, 是在无线端替换jquery,zepto,seajs,angulars,task任务处理,mvc等的理想框架.

async

- wind.js

promise

- when
- q
- Bluebird: 这个类库除了兼容 Promise 规范之外，还扩展了取消promise对象的运行，取得promise的运行进度，以及错误处理的扩展检测等非常丰富的功能，此外它在实现上还在性能问题下了很大的功夫。

语言识别

- julius.js

iframe相关

- iframe-resizer: Keep same and cross domain iFrames sized to their content with support for window/content resizing, in page links, nesting and multiple iFrames. (Dependency free, IE8+)

图表

- metricsgraphics.js

SVG

- SVG Morpheus
- Vivus
- paper.js: 强大的矢量图形绘制库。
- awesome-svg: A curated list of SVG.

动画相关

- particles-js
- Four Shadows: 添加扁平化阴影
- close-pixelate: 使图像像素化
- zynga/scroller: A pure logic component for scrolling/zooming. It is independent of any specific kind of rendering or event system.
- nprogress: 进度条插件。
- Headroom.js: Give your pages some headroom. Hide your header until you need it.

视差滚动

- skrollr: Stand-alone parallax scrolling library: 全平台视差效果插件，纯JS，无依赖。

功能性能

- zip.js
- Screenful.js: 使得任何元素都可以全屏
- Binoculars: 数据捕获的库。
- draggabilly: Make that shiz draggable.
- sweetalert: A beautiful replacement for JavaScript's "alert".

物理引擎

- Matter.js: Matter.js is a 2D physics engine for the web.

测试相关

- Atomus

参考资料

- 2014年12月最棒的 15 个 JavaScript 库
- 前端常用插件汇总
- 前端组件库: 搭建web app常用的样式/组件等收集列表(移动优先)
- 2015 JavaScript Frameworks in the Real World

CSS相关库

前端框架

- Bootstrap
- Primer: github的css框架
- purple: heroku的css框架

Bootstrap资料

代码片段

- Bootsnipp: Code snippets for Bootstrap.
- TODC Bootstrap: A Google-styled theme for Bootstrap.

一些免费的主题

- ShapeBootstrap
- Bootstrap Zero
- Start Bootstrap

参考资料

移动开发

css框架

- Bootstrap
- Foundation
- Semantic UI
- Pure.css
- uikit

js库

- app.js
- touchstone.js: React.js powered UI framework for developing beautiful hybrid mobile apps.
- frozenui: Frozen UI是一个开源的简单易用，轻量快捷的移动端UI框架。基于手Q样式规范，选取最常用的组件，做成手Q公用离线包减少请求，升级方式友好，文档完善，目前全面应用在腾讯手Q增值业务中。
- Amaze: 中国首个开源 HTML5 跨屏前端框架.

参考资料

- The 5 Most Popular Frontend Frameworks of 2014 Compared

Loading动画

CSS实现

- sass实现圆柱体加载动画
- sass实现滚轮加载动画
- css3实现扁平风格加载动画1
- css3实现扁平风格加载动画2

辅助工具

- 代码运行可视化: 可查看出运行时候的作用域。
- JS Comparison Table: if, ==, ===各种对比情况
- JSDB: The definitive source of the best JavaScript libraries, frameworks, and plugins.
- JavascriptOO: every javascript project you should be looking into.
- Favicon Generator: Generate favicons for PC, Android, iOS, Windows 8.
- UI Box: 收集UI相关的代码。

性能优化

- InstantClick: Makes your pages load instantly by pre-loading them on mouse hover.

动画

- Hover: css3实现动画效果
- AniCollection: 也是css3实现的按钮动画

正则相关

- 可视化正则表达式